

PROGRAM AND DATA STRUCTURES IN PL 7044

A Thesis Submitted
In Partial Fulfilment of the Requirements
For the Degree of
MASTER OF TECHNOLOGY

by

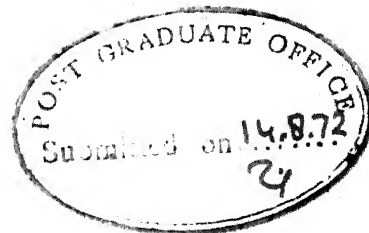
ASHOK GUPTA

to the

Department of Electrical Engineering
Indian Institute of Technology, Kanpur

August, 1972

TH
EE/1972/M
6959b



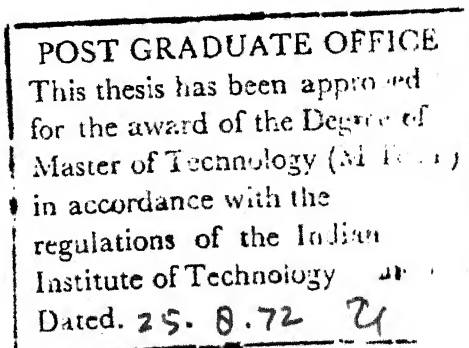
CERTIFICATE

This is to certify that the thesis entitled "Program and Data Structures in PL 7044" is a record of work done by Ashok Gupta under my supervision, and it has not been submitted elsewhere for a degree.

H. V. Sahasrabuddhe

H.V. Sahasrabuddhe
Assistant Professor
Electrical Engineering Department
Indian Institute of Technology, Kanpur.

Kanpur
August 12, 1972



V
JUNE '76

29 SEP 1972

I. I. T. KANPUR
CENTRAL LIBRARY
Acc. No. A 21157

thesis
001.6424
G959

EE-1972-M-GVP-PRO

-TO MY MOTHER

ACKNOWLEDGEMENTS

With gratitude, I express my thanks to Dr. Hari V. Sahasrabuddhe, Assistant Professor, Department of Electrical Engineering, for his exemplary guidance and constant encouragement throughout the course of this work. It was indeed a pleasure and privilege to have worked under his guidance.

I would like to take this opportunity to thank my colleagues Mr. K. Kannan and Mr. A.B. Saha for the countless number of discussions we have had in connection with this project. But for their sustained cooperation, it would not have been possible to face the innumerable hurdles on our path.

My thanks are also due to Mr. Vipin K. Shrivastava, who drew the large number of flow charts and figures appearing in this volume.

Mr. J.K. Misra's excellent typing of the voluminous manuscripts deserves special mention.

Finally, I thank all my friends who have made my stay here a pleasant one.

Ashok Gupta

IIT Kanpur-16

August, 1972

PREFACE

This thesis and other reports, namely, 'Input and Output Facilities in PL 7044' and 'An Expression Processor for PL 7044' describe in detail a compiler writing project for the language PL/I on the System IBM 7044. It has been the endeavour of the authors to present as much information about the various aspects of the project such as formulation, coding, debugging methods etc. as is possible in a report of this nature.

We have tried to cater to basically two different types of readers: the casual reader who is interested in a bird's eye view of the project but not in details and the serious reader who is planning to embark on a similar project. It is in satisfying the demands of the latter category of readers that this report may be found wanting by atleast some of them, because it is extremely difficult to recognise when one has given sufficient details about a particular strategy adopted and at the same time ensure that no redundancy has crept in. Also, since the project has been described in three separate volumes which are not necessarily logically independent, a certain amount of incoherency may be discernable. We have tried to solve the first problem by illustrating all important aspects by suitable examples and how the compiler goes about decoding them into MAP instructions. The second problem has been partially overcome by providing a large number of Appendices at the end of each of the theses.

PREFACE

This thesis and other reports, namely, 'Input and Output Facilities in PL 7044' and 'An Expression Processor for PL 7044' describe in detail a compiler writing project for the language PL/I on the System IBM 7044. It has been the endeavour of the authors to present as much information about the various aspects of the project such as formulation, coding, debugging methods etc. as is possible in a report of this nature.

We have tried to cater to basically two different types of readers: the casual reader who is interested in a bird's eye view of the project but not in details and the serious reader who is planning to embark on a similar project. It is in satisfying the demands of the latter category of readers that this report may be found wanting by atleast some of them, because it is extremely difficult to recognise when one has given sufficient details about a particular strategy adopted and at the same time ensure that no redundancy has crept in. Also, since the project has been described in three separate volumes which are not necessarily logically independent, a certain amount of incoherency may be discernable. We have tried to solve the first problem by illustrating all important aspects by suitable examples and how the compiler goes about decoding them into MAP instructions. The second problem has been partially overcome by providing a large number of Appendices at the end of each of the theses.

To enumerate systematically, Chapter I is an introduction to the language itself. We have obviously refrained from cataloguing all of the features of the language for this will entail a separate book by itself; we have instead highlighted those features which give the language the beauty, the flexibility, the power that is lacking in other high-level languages. PL 7044, which is a fairly large subset of PL/I is described in some detail in Chapter II. [Here the emphasis is on the selection of the subset, the factors that influenced us in the selection and some interesting techniques adopted to implement powerful features incorporated in PL 7044.] Chapter III is a description of the First Pass Processor while Chapter IV contains a description of declarations, symbol table and attributes analysis routines. Chapter V deals with all the program control statements of PL 70 except for the DO-statement, which is described in the report titled 'Input and Output Facilities in PL 7044'.

✓ IBMs PL/I Language Specifications manual, Order No. GY33-6003-2, is the parent source of the Syntax notation, General Format, Syntax Rules and General Rules for various statements.

At the time of writing this report, coding and debugging of individual routines comprising the compiler has been completed, although test running the routines together could not be carried out due to unavoidable circumstances.

CONTENTS

<u>Chapter</u>		<u>Page</u>
I	Overview of the PL/I Language	OP-1
1.	PL/I: An Introduction	OP-1
1.1	Why PL/I	OP-1
1.2	PL/I: Its Philosophy	OP-2
2.	PL/I: Some Features	OP-4
2.1	Program Structures	OP-4
2.2	Data Structures	OP-12
2.3	Conversions	OP-22
2.4	Storage Classes in PL/I	OP-24
2.5	Data Transmission	OP-25
2.6	Asynchronous Operations in PL/I	OP-27
2.7	Compile Time Statements	OP-30
3.	PL/I and List Processing	OP-33
II	Overview of the PL 7044 Compiler	OC-1
1.	The Choice of a Subset	OC-1
2.	Organisation of the PL 7044 Compiler	OC-2
2.1	Choice of Number of Passes	OC-2
2.2	Pass I	OC-4
2.3	Pass II	OC-6
2.4	Pass III	OC-7
3.	Implementation Details	OC-9
3.1	Memory Management	OC-9
3.2	Expression Processor	OC-26
3.3	Input and Output	OC-32
3.4	Concluding Remarks	OC-44

<u>Chapter</u>		<u>Page</u>
III	First Pass Processor	FP-1
	1. First Pass Activities	FP-1
	2. Description of First Pass Activities	FP-2
	2.1 Detection of Complex Constants	FP-3
	2.2 Detection of '*' Used as an Operand	FP-3
	2.3 Detection of 'iSUB' Variables	FP-4
	2.4 Detection and Application of Repetition Factors	FP-4
	2.5 Collection of Statement Labels	FP-5
	2.6 Making a Name Table	FP-14
	2.7 Stack Mechanism	FP-16
	2.8 Classification of Statements	FP-18
	2.9 Matching IF-THEN-ELSE	FP-20
	2.10 Classification of END and Multiple Closure	FP-29
	2.11 DO Predecessor Table and Block Predecessor Table	FP-30
	2.12 Separation of Specification and Declarations from the Text	FP-32
IV	Analysis of Declarations and Symbol Table Management	DC-1
	1. Declarations and Declare Statements	DC-1
	1.1 Declarations	DC-1
	1.2 Declare Statement	DC-2
	1.3 Organisation	DC-4
	1.4 Description of Modules	DC-5

Chapter

Page

2.	Symbol Table	DC-12
2.1	Introduction	DC-12
2.2	Organisation	DC-13
2.3	Service Routines of Symbol Table	DC-15
2.4	Symbol Table Format	DC-18
3.	Attribute Analysis Routines	DC-25
3.1	Introduction	DC-25
3.2	Classes of Attributes	DC-26
3.3	Attribute Analysis Routines	DC-26
3.4	Main Module	DC-28
3.5	Analysis of INITIAL Attribute	DC-42
V	Program Control Statements	PC-1
1.	IF Statement	PC-1
1.1	Syntax	PC-1
1.2	Coding of IF-Statement	PC-2
2.	GOTO Statement	PC-4
2.1	General Format	PC-4
2.2	General Rules	PC-4
2.3	Examples and Implementation	PC-5
3.	Function Reference and Call Statement	PC-13
3.1	Syntax	PC-13
3.2	RETURN Statement	PC-15
3.3	BEGIN Statement	PC-18
3.4	END Statement	PC-18

ChapterPage

4.	PROCEDURE and ENTRY Statement	PC-20
4.1	Syntax	PC-20
4.2	Prologue and Parameter List	PC-22
4.3	Coding of PROCEDURE and ENTRY Statements	PC-28

LIST OF APPENDICES

	<u>Page</u>
A1 Character Set	AP-1
A2 Lexical Representation	AP-3
B Constant in PL 7044	AP-6
C Names in PL 7044	AP-12
D1 Variable Types in PL 7044	AP-14
D2 Major Types and Minor Types	AP-20
D3 Symbol Table Representation of Major Types	AP-22
E Program Organised to Explain Scope of Declarations etc.	AP-28
F First Order Storage Assignment	AP-32
G1 Attributes of Files	AP-36
G2 Default Attributes	AP-36
H Built-in Procedures and Pseudo Variables	AP-38
I1 Statements allowed in PL 7044	AP-43
I2 Attributes Allowed in PL 7044	AP-44
I3 Logical and Storage Restrictions	AP-46
J1 PL 7044 Program Listing Format	AP-50
J2 Error Recovery	AP-51
K Format of File Status Block	AP-52
L1 Buffer Handler Output	AP-55
L2 Expression Processor Output (Before Sorting)	AP-57
L3 Expression Processor Output (After Sorting)	AP-62
L4 MAP Code Generated	AP-64

	<u>Page</u>
M1 Coding of Edit Directed Output Statement	AP-71
M2 Coding of DO Statement Outside an I/O Statement	AP-76
M3 Input and Output Formats	AP-78
N Output of Format Generator	AP-84
Ø Run Time Stack Management	AP-85
P Free List	AP-93
Q PL 7044 Compiler Output	AP-96
R Glossary of Important Words and Terms	AP-101
Bibliography	AP-106

SYNTAX NOTATION USED

1. Braces $\{ \}$ are used, to denote grouping. Both vertical stacking of syntactical units and vertical stroke have been used to indicate that a choice is to be made e.g. both

$$\text{identifier} \begin{Bmatrix} \text{FIXED} \\ \text{FLOAT} \end{Bmatrix}$$

and identifier $\{ \text{FIXED} | \text{FLOAT} \}$

have same meaning, i.e. 'identifier' must be followed by the literal occurrence of either the word `FIXED` or the word `FLOAT` but not both.

2. Square brackets $[]$ denote options. Anything enclosed in brackets may appear once or may not appear at all.
3. Three dots \dots denote the occurrence of immediately preceding syntactical unit one or more times in succession.
4. Lower case letters are used to show nonterminal (syntactic) variables where as only CAPITAL letters are used to construct a terminal variable.

CHAPTER I

OVERVIEW OF THE PL/I LANGUAGE

1. PL/I: An Introduction:

1.1 Why PL/I:

The rapid growth of the computing machinery during the past two decades has necessitated a constant re-evaluation of the means by which the digital computer could be put to efficient use in the service of mankind. This is especially true in the field of language development. A consequence of this development has been PL/I - a language designed and promoted by IBM. Today PL/I is a fairly well established language and is fast gaining currency among users of IBM as well non-IBM equipment. Chief among the reasons for its success has been the fact that IBM have gone to great lengths in making the language versatile and useful to a wide range of computer users and at the same time making each class of users feel that the language has been tailored to its needs. This is not to say that PL/I has no shortcomings. To date, PL/I has not been implemented in toto on any digital machine and this is reflective of the practical difficulties in writing efficient compilers for PL/I. However, fairly large subsets have been introduced on IBM machines and whatever may be ones reservations about it, one has to learn to use PL/I if one wishes to take advantage of the large number of scientific and commercial programs written in PL/I. A logical

extension of this necessity is the availability of a PL/I compiler within ones reach. To provide such a compiler to the users of IBM 7044 has been the goal of Project PL/7044.

1.2 PL/I: Its Philosophy:

In order to appreciate how PL/I is different from other high level languages it is necessary to examine the goals of the designers. These were a) to serve the needs of an unusually wide spectrum of users - scientific, commercial, real time and systems programmers - and to allow both the novice and the expert to find facilities at his own level b) to design the language such that programs could be written in accordance with the natural description of the problem it is supposed to solve so as to minimise coding errors and c) to provide a language for implementing present and future generations of compilers, monitors and the like - the question here being not so much of 'Can we write PL/I in PL/I' (we certainly can) but of 'Can we write in PL/I a real time operating system to support PL/I program'.

Not all of the above goals have been achieved in PL/I nor can it be said that any one of them has been completely achieved, but it does equip the user with tools that are not available in any other high level language.

In aiming for the goals enunciated above, certain principles were laid down which formed the framework of the new language and every feature of the language was motivated by one or more of these principles.

These principles were

- (i) Anything goes: The new language PL/I should have a liberal semantic structure. If a reasonable interpretation can be given to a particular combination of symbols then it should be allowed per se by the language. A PL/I compiler should not have too many permissive diagnostics ("This is wrong but I am smart enough to know what you really mean"), on the other hand, it should have warning messages ("Do you really want this to be done in this strange way"). Such messages will enable compatibility among different implementations.
- (ii) Full access to machine and system facility: This is a particularly useful guideline in so far as systems programmers are concerned but very difficult to achieve in practice.
- (iii) Modularity: "If you need it, it is there, but if you don't, you don't have to know about its existence". This was the motivation behind giving every name, every option, every facility a default interpretation which is so chosen that it is likely to be the one required by the user who is unaware that alternatives exist. However, one has to pay a penalty for this kind of a design, for if one forgets to assign an attribute other than the default attribute, no diagnostics will be provided.
- (iv) Relative machine independence: Although this is a sound principle in itself, it is also one that most formulators tend to overlook. The makers of the PL/I language have achieved a measure of success in this area but even then PL/I is not

altogether free of this shortcoming. In fact it does tend to reflect the organisation of the IBM 360 family of computers, particularly in the field of data structures.

(v) Catering to the novice: More than one method of specifying a particular problem is made available to cater to the needs of the novice. This will enable the user to use the notations most familiar to him and at the same time allow the compiler to maximise efficiency of the most frequently used case.

(vi) Choice of PL/I formats: A free format has been chosen to facilitate ease of coding and punching. 48 and 60 character sets are provided so as to allow implementation on different machines.

2. PL/I: Some Features:

For obvious reasons, it is neither desirable nor feasible to catalogue the large repertoire of PL/I features that have distinguished the language from other high level languages. However, we shall attempt to highlight a few salient features that are mainly responsible for its enhanced status.

2.1 Program Structure:

2.1.1 Basic Language Elements:

As already mentioned PL/I allows the program to be written in a free field format. The basic element is the statement which is separated from another statement by the semicolon and is a combination of characters (from either the 48 or 60 character set). Thus a program is simply a string of characters with no internal grouping.

2.1.2 Groups:

An important aspect of PL/I program structure is the entity called GROUP. Quite often, in programming, we are faced with a decision structure where the alternatives require the execution of more than one statement. In FORTRAN, either we use a subroutine or branch off to some other part of the program and then come back to the point from where we branched off. Both these methods are unsatisfactory the first because one may have to prepare the subroutine in the sense of providing it with the right type of arguments, initialization etc., while the second can become very clumsy if the decision mechanism requires nesting of such groups. The 'DO' statement in PL/I can handle such situations elegantly as the following example illustrates

FORTRAN

```

      IF(A.EQ.B) GOTO 10
      B=B-1
      A=A+1
      PRINT 5
5     FORMAT (* A AND B ARE NOT EQUAL *)
      GOTO 25
10    READ 15,C
15    FORMAT (Ø12)
      D=A*SQRT(C)
      PRINT 20,A,B,D
20    FORMAT (3Ø12)
25    CONTINUE

```

PL/I

```

IF A=B THEN DO;
    GETLIST(C); D=A*SQRT(C);PUT DATA (B,C,D);
    END;
ELSE DO;
    B=B-1;A=A+1; PUT LIST('A AND B NOT EQUAL');
    END;

```

Notice that PL/I program does not contain any GOTO's, while the FORTRAN program has 2.

The logical extension of the DO groups is in their use in repetitive calculation. In this respect also, the PL/I DO statement turns out to be more powerful than its counter part in FORTRAN. For example, consider

	FORTRAN	PL/I
	DO 10 I=1,200	DO:DO P=1.TO 100 BY 0.5
	IF(A.GE.B)GOTO 20	WHILE A < B;
	P=I	.
	P=P/2.0	.
	.	.
	.	END DO;
10	CONTINUE	
20	

Notice that a single statement in PL/I is equal to as many as 4 in FORTRAN in the above example.

2.1.3 Compound Statements:

Of the two compound statements, namely the IF and ON statement we shall discuss the more important IF statement. The first example of the preceding section already contains a sample use of the IF statement. IF clauses can also be nested as shown in the example below:

```
IF A>B THEN IF C<D THEN IF T THEN C=0;
ELSE D=0; ELSE IF B<F THEN K=0; ELSE B=0;
ELSE IF P+Q<M THEN M=P+Q; ELSE T=0;
```

The equivalent FORTRAN program would be as follows:

```

      IF(A.LE.B)GOTO 1
      IF(C.GE.B)GOTO 2
      IF(T)GOTO 3
      D=O GOTO 4
      GOTO 4
3     C=O
      GOTO 4
2     IF(B.GE.F) GOTO 5
      K=O
      GOTO 4
5     B=O
      GOTO 4
1     IF((P+Q).GE.M) GOTO 6
      M=P+Q
      GOTO 4
6     T=O
4     CONTINUE

```

2.1.4 Procedures and Blocks: In the hierarchy of PL/I program structure, procedures and blocks occupy the top positions. In programming complex problems, it often becomes necessary to execute a set of statements repeatedly with different sets of input data. Also it becomes convenient to delimit the scope of applicability or uniqueness of a name so that names may be non-unique within a program and yet be well defined locally. These two requirements have been adequately met in PL/I with the introduction of the block and procedure features. We shall first discuss the concept of the Procedure and then that of the block.

Each procedure is surrounded by its own header and trailer statements serving to identify the body of the procedure to the compiler. The label(s) associated with the procedure header is/are the one(s) by which it is referenced and is/are termed the primary entry point(s). A procedure may also have several

secondary entry points from where execution may be continued. Procedures are never executed in sequence i.e. they must be invoked by a specific occurrence of the name of the procedure. All names encountered within the body of the procedure are not 'known' outside the procedure except when they are given the attribute EXTERNAL. Procedures may be stacked one after another, each forming a separate entity by itself or they may be nested. In the former case, they are known as external procedures while in the latter case they are known as internal procedures. External procedures are similar to FORTRAN subroutines in the sense that they can be compiled separately and used with different programs. But the similarity cannot be carried much further because, as we shall see presently, the procedure concept is a much more powerful concept than the subroutine feature. External procedures may be invoked from any point in a program whereas internal procedures can be invoked only if the immediate predecessor is active. Further, the procedure may be classified as a subroutine procedure or a function procedure depending on whether the referencing is by a call statement or by its occurrence as an operand in an expression.

The 'block', also called a 'begin block', is mainly used for delimiting the scope of names in a program. They are structurally similar to procedures but cannot be invoked from out-of-line and can be entered strictly in sequence or by a GOTO statement whose destination is the label prefixed to the block.

Like the procedure the block has its own header and trailer statements.

The block and procedure form a powerful set of tools in the hands of the PL/I programmer. But one has to exercise caution in the manner in which the block or procedure is invoked in order to ensure that proper transfer of information takes place at the time control is transferred from one procedure to another.

Both the parameter list and the argument list should normally be accompanied by what are known as specifications which assign attributes to the arguments and parameters. If specifications are not provided, PL/I will assign default attributes, but in many cases the default mechanism may be inadequate and may lead to error conditions. For example, consider:

```
P:PROCEDURE (A,B,C);  
  .  
  .  
  .  
END P;
```

Since nothing has been specified about A,B,C they will be assigned default attributes in accordance with the rules of PL/I. When P is called from some other point, the attributes of the argument are, in general, not checked, unless explicitly told to so, to determine if they match those of the corresponding parameters of P. If checking is done, and if there is a mismatch, then conversions will be attempted which may not be

always successful. It is safe, therefore, to ensure that either the attributes of the arguments and parameters match or if conversion is to be allowed, explicit directives are given to the compiler so that conversion takes place along predictable lines. When arithmetic expressions are used as arguments, it is likely that the attributes of the result do not match those of the parameters of the called procedure. In order to avoid unpredictable conversions, PL/I allows the programmer to issue directives to the compiler in what is known as an ENTRY list which specifies the attributes of the argument expected by the called procedure. For example, consider,

```

Q:PROCEDURE(M,N,L);
  DECLARE M CHARACTER, N FIXED, L BIT;
  DECLARE P ENTRY (FIXED (5,10), FLOAT (8,4),
                  CHARACTER (5));
  .
  .
  CALL P(M,N,L);
  .
  .
  CALL P((M*N+L), M/N, (L+M+N));
  .
  .
END Q;

```

In this case the entry list gives a complete idea of what P expects and therefore at the time P is called, the conversion will be performed, if necessary, to the respective targets. This avoids unpredictable results later.

Procedures may be referenced recursively by assigning the RECURSIVE attribute to the procedure name. The following procedure which can be used for calculating the factorial of an integer, is an example of a recursive attribute.

```
FACT: PROCEDURE(N) RECURSIVE;
      IF N=1 THEN RETURN (1);
      RETURN (FACT (N-1)*N);
      END FACT;
```

Let us wind up our discussion on procedures by describing by what is definitely one of the most powerful features of the language, namely the GENERIC attribute. Sometimes it is necessary to have a set of procedures each of which is to be invoked for each of the several types of arguments. Trigonometric functions operate in this manner. Also the procedure to be invoked may depend on the number of arguments. There can be a function procedure which evaluates the roots of a polynomial. The coefficients of the polynomial are to be supplied as parameters. Now for polynomials of 1st, 2nd and 3rd order, simple formulae are available. Thus, for example,

```
DECLARE SIN GENERIC (SIN1 WHEN (REAL), SIN2 WHEN (COMPLEX));
DECLARE ROOTS GENERIC (ROOT1 WHEN (*,*), ROOT2 WHEN(*,*,*),
                      ROOT3 WHEN (*,*,*,*));
```

The WHEN clause determines which alternative is to be selected when the generic family name is referenced.

2.2 Data Structures:

2.2.1 Data Organisation:

Any information that is manipulated by a program may be termed 'DATA'. Data is organisationally divided in PL/I into scalar data items and aggregates of data items.

Scalar data may be either constants or variables as in any high level language. As far as arithmetic constants are concerned, there is not much difference basically between PL/I and FORTRAN except that in PL/I one can direct the compiler to store the constant in a particular fashion (i.e. DECIMAL or BINARY format, FIXED or FLOATING representation etc.). A generalized example would be $-10.2367 \text{ F-03Y}(8,2)\text{I}$, which means that the Fortran constant -10.2367×10^{-3} must be scaled by a factor of 10^2 , represented to 8 positions of precision in binary form and is to be treated as an imaginary constant.

Scalar variable items are also stored in accordance with the scale, mode and precision attributes assigned to them.

Data aggregates may either be arrays or structures or as extended array of structures or structure of arrays. While Fortran users are familiar with the notion of an array, the structure concept is unique to PL/I and is not available in Fortran or Algol. Although some structuring is allowed in COBOL, it is relatively primitive.

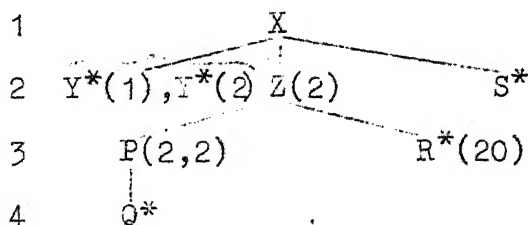
A major difference between FORTRAN arrays & PL/I arrays is the manner in which they are stored. In PL/I arrays are stored in row major order. Although this difference is unlikely to affect the normal PL/I user, a systems programmer should take note of this difference while manipulating arrays.

A structure in PL/I is a collection of data items among which a certain hierarchy is maintained. The data items will in general belong to different data types. It is evident that structures may be 'contained' in structures. In other words a structure may be an element of another structure. In such cases, the parent structure is called the major structure while all other structures are called minor structures. An element of a structure may be an array whose elements themselves may be structures. A generalized example of a structure and the equivalent tree is given below:

```

DECLARE 1 X,
        2 Y(2),
        2 Z(2),
            3 P(2,2),
                4 Q FLOAT,
            3 R(20),
        2 S CHARACTER(5);

```



* Base Elements.

Note that leaves or base elements as they are called, can have attributes appended to them. Base elements should normally use qualifiers in order to make unambiguous references. Cross

Cross Section of Arrays:

The concept of cross-section of arrays is a logical extension of the subscripting notation. Its meaning and use are best illustrated by an example,

$$A(I,J,K,*) = A(J,*,J,K) + A(*,I,J,K);$$

In the above statement, the left hand side is said to be a one dimensional cross-section of the 4 dimensioned array A.

Its elements range from $A(I,J,K,N)$ to $A(I,J,K,M)$ where N and M are the current lower and upper bounds of the 4th subscript. The dimensionality of the cross-section is determined by the number of asterisks appearing in the subscript list.

While we are dwelling on the subject of arrays and structures, it is worth while to discuss three features of PL/I which facilitate a high degree of manipulation with data aggregates. These are the DEFINED, LIKE and BY NAME attributes.

DEFINED Attribute:

Although the DEFINED attribute may be used with scalar data items also, it is more useful in the case of data aggregates. Quite often, programming situations demand that we be able to refer to a piece of data by more than one name. This is done by

declaring the variable in question in the usual manner and then following it up with a 'Defined' attribute and the name of the variable with which our original variable is to be synonymous.

Consider the statement `DECLARE A(10,10), B(10,10) DEFINED A;` this will cause the compiler to allocate space for A but for B it will not allocate any space. Each reference to B will be treated as a reference to A. Thus A is the defining variable while B is the defined variable. It is not necessary for B to have the same dimensions of A. For example one can write `DECLARE A(4,5), B(3) DEFINED A(1,1);` which will result in the following equivalences,

$$[(A(1,1), B(1)), (A(1,2), B(2)), (A(1,3), B(3))]$$

There are a few restrictions on the declaration of B and A. For instance B should have the same attributes as A, A itself cannot be a cross-section of some array, and A itself cannot be defined in terms of some other array.

In cases where the dimensions of the two arrays involved in a DEFINED attribute, do not match, care should be exercised in avoiding extension of the defined array beyond the bounds of the defining array.

Even more complicated relationships may be set-up between the defined and defining array with the use of 'iSUB' feature. This is best illustrated by an example,

```
DECLARE A(20), B(10) DEFINED A(2*iSUB-1);
```

We calculate the relationship between A and B as follows:

$$\begin{aligned} B(1) &\equiv A(2*1-1) \equiv A(1), & B(2) &\equiv A(2*2-1) \equiv A(3), \\ B(3) &\equiv A(2*3-1) \equiv A(5), & B(4) &\equiv A(2*4-1) \equiv A(7) \end{aligned}$$

and so on.

The expression inside the paranthesis specifying the relationship can be any scalar expression in 'iSUB' where 'iSUB' denotes the value of the i-th subscript in the defined array, i taking values from 1 to n when n is the number of dimensions of the defined array. One can easily visualize how this feature can be put to use in several programming situations. For instance, one can selectively operate upon a matrix to obtain certain results, or one can use the 'iSUB' expression as a sort of code which when used to select elements in a matrix will reveal some intelligible information. A person who does not know the code cannot extract the information although he may have access to the matrix as a whole. This is particularly useful in a time sharing environment where many customers may have access to certain files.

Just as 'defined' attribute could be applied to arrays, so can they be used with structures. Entire structure or part of structures may be given alternative names by the use of the DEFINED attribute.

Consider the example,

```
DECLARE 1 A, 2 B CHAR(4), 2 C CHAR (5), 2 D CHAR (8);
DECLARE E CHAR (17) DEFINED A;
```

When reference is made to E, the 17 characters allotted to the structure A will be involved. Here we see that a byte oriented machine has been implicitly assumed although the design guide lines call for complete machine independence. Thus features such as these would be difficult to implement in a word oriented machine such as the IBM 7044.

LIKE Attribute:

Frequently, we find that two structures have the same hierarchy and the elements have the same attributes. In such cases it would be tedious to declare the two structures separately and in full especially when the two structures have many minor structures and base elements. With the LIKE attribute it is possible to use a short hand notation for all but the first structure so declared. A very good example * would be,

```

DECLARE 1 SOAP, 2 NAME, 3 TRADE CHAR (5),
          3 TECH CHAR (8),
          2 DATE, 3 MONTHS FIXED (2),
          3 DAY FIXED (2),
          3 YEAR, 4 DECADE FIXED (1),
          2 INVENTORY, 3 LARGE, 4 BOXWT FIXED (5),
          4 BOXES FIXED (6),
          3 GIANT, 4 BOXWT FIXED (5),
          4 BOXES FIXED (6),
          3 KING, 4 BOXWT FIXED (2),
          4 BOXES FIXED (6);

```

Now if we had another product, say CAKE_MIX which had similar structure, we could simply write,

```

DECLARE 1 CAKE_MIX LIKE SOAP;

```

Even if only a part of a structure matches with the structure in hand, we could still use the LIKE attribute.

Just as in the case of the DEFINED attribute the LIKE attribute cannot be carried over more than one level. Thus in the above example SOAP itself cannot be 'likened' to some another structure.

BY NAME Feature:

Manipulation with structures can be made selective with the help of the BY NAME feature. In this case manipulation on the set of base elements will take place only if at every level from the base upto the root (excluding the root name) the names of the nodes match. As an example consider,

```
DECLARE 1 A, 2 A1, 2 B1, 3 D1, 2 C1, 3 K1,
        1 B, 2 C1, 3 K1, 3 B1, 2 A1, 2 B1,
        1 C, 2 A1, 2 C1, 3 K1, 3 B1, 4 B1;
```

Now, $A = B + C$ BY NAME; would result in

$A.A1 = B.A1 + C.A1$; and

$A.C1.K1 = B.C1.K1 + C.C1.K1$;

2.2.2 Data Types:

We have so far discussed data on the basis of their organisation in the computer and a few related attributes. We shall now turn our attention to data classified according how a PL/I program operates upon them. Under this classification we can talk of Problem data types and Program control **data** types.

Problem data consists of Arithmetic and string data types. Every data item which is of the arithmetic type is

is assigned a base (or radix of 2 or 10), mode (real or complex) and a scale (fixed point or floating point) and a precision which is machine dependent. As has been stressed earlier, these attributes, can either be explicitly specified by a declaration or they will be assigned default interpretations. We can also specify a PICTURE attribute in which a numerical interpretation will be given to data stored in character or bit string format as opposed to coded format of non Picture arithmetic Data described earlier. A data item which has the PICTURE attribute is complete in itself and does not require any other attributes of mode, scale etc. to be assigned to it. The motivation behind the provision of PICTURE attribute is that commercial computation requires not only considerable editing, but also of late, substantial amount of numerical calculations. This is one of the facts overlooked by designers of other higher level languages. Algol and Fortran, which have powerful arithmetic facilities have poor editing characteristics while COBOL has good editing facilities but rigid arithmetic. It is appropriate to mention that whereas in PL/I attributes are listed for each variable or for groups of variables, in Fortran and Algol, declarations are made according to data types.

FORTRAN

```
INTEGER   A,B,C
DIMENSION A(10),B(10,10)
REAL     I
COMMON   /COM/I
```


PL/I

```
DECLARE (A(10), B(10,10), C) FIXED (8,0),
        I FLOAT (8,5) EXTERNAL;
```

String Data may be classified as character strings or bit strings. String data are characterised by their length which may be fixed or varying during a program. While character strings usually find their use in editing results of arithmetic computation, bit strings play an important role in performing logical decisions in a PL/I programs. Consider, for example the ALGOL program

```
A: = IF C THEN B ELSE D;
```

In PL/I the same program may be written as,

```
A = B*C + D* (NOT C);
```

Here C is a bit string of length 1. Another important use of bit string in PL/I is for systems programmers. Languages like Fortran and Algol have no bit manipulation capabilities per se. There are many built in functions in PL/I for string data which allow considerable manipulation and editing operations without having to resort to unnatural methods as in FORTRAN.

Let us briefly discuss the other major data type, namely the Program Control Data Types. Any data that can be classified as label, entry, task, event, pointer, offset or area type can be regarded as Program Control Data. We have already discussed entry data type under program structure. POINTER, OFFSET and

AREA variables are discussed under 'List Processing facilities' while TASK and EVENT are discussed under 'Asynchronous Operations' in this Chapter.

Label data items may be constants or variables. Label constants are used to identify statements (similar to statement numbers in FORTRAN) while label variables which could be arrays or scalars are used to perform program controlled transfers. A label variable has, at any stage in the program, as its value a label constant which may then be used as the destination of a GOTO statement. These label variable can be used to perform a function similar to computed GOTO in Fortran, although they (label variables) are much more versatile than just computed GOTOs. For example,

```

MAIN:  PROCEDURE OPTIONS (MAIN);
        DECLARE (L, ML)(3) LABEL, (M,N) INITIAL (1);
        L:  I=(M+3*N)/3
            L(I)=ML(N); GOTO L(I);
            .
            .
ML(1):  M=0; N=1;
        .
        .
        GOTO L;
ML(2):  M=-3; N=3;
        .
        .
        GOTO L;
ML(3):  M=3; N=2;
        .
        .
        GOTO L;
        END MAIN;

```

2.3 Conversions:

Consistent with the philosophy of 'Anything Goes', PL/I performs a wide range of conversions in assignment statements, I/O statements, procedure calls etc. It is impossible to cover all the situations where conversions take place, and hence we shall limit our discussion to certain basic principles underlying conversions.

In arithmetic expressions, operations are always performed on coded arithmetic data only. If any of the operands is not so coded, it is first converted to coded form before proceeding with the evaluation of the expression.

Effect of Conversion on Precision:

If the precision of the operands in an infix operation differs, no conversion will be done. In case of floating point operations the result will have the precision of the greater of the precision of the two operands. In case of fixed point operations, precision of the result depends on the type of operations and the precisions of the two operands. In every case any necessary truncation will always be made towards zero.

Mode Conversion:

Complex items when converted to real mode, lose their imaginary part and only the real part is retained. When real items are converted to complex items, the result has a zero as the imaginary part. In mixed mode operation any real item will be converted to complex item.

Scale and Base Conversion:

The conversions here are fairly involved and no generalization can be made. For particular cases, the specification manual* should be consulted. Conversions may also take place between arithmetic and string types of data or between character and bit string types.

When bit strings are converted to character type then each bit is converted to the corresponding character and the result has a length equal to the original bit string length.

When character strings are converted to bit string the character string must contain only 0's and 1's, otherwise a conversion error will be raised.

When character strings are converted to arithmetic type, they may contain only those characters which can be coded into arithmetic form.

When bit strings are converted into arithmetic form they are interpreted as though they were unsigned binary integers.

Arithmetic to character strings have no special properties and follow the rules of list directed I/O. When arithmetic data is to be converted to bit string, they are first converted to a real binary fixed point number. The bit string thus resulting, is left justified and has a length equal to the precision of the binary number into which the arithmetic item was first converted.

*PL/I Language Specifications (IBM) Order Number GY33-6003-2

2.4 Storage Classes in PL/I:

Four classes of storage are possible in PL/I.

The **AUTOMATIC** attribute serves to denote the class of storage where the various data items are assigned storage only when the procedure in which they are declared is invoked. This type of storage is dependent on the logic of the program and is the normal or default class assigned to variables whose scope is **INTERNAL** or whose scope is unspecified.

The **'STATIC'** attribute when assigned to a variable causes allocation of storage at compile time itself and such a data item occupies fixed locations in memory at all times during the execution of a program. This is similar to the storage allocation of Fortran variables and Algol's **OWN** variables. If a data item is **EXTERNAL** in scope then it is assumed to have the **'STATIC'** storage class if no other storage class is specified.

The third type of storage class allows a programmer to allocate and free storage space according to his choice. Repeated allocations of such data items cause previous **'generations'** to be pushed down so that the value of the variable available for manipulation is the one corresponding to the last invocation. Similarly the storage for a variable may be released or **'freed'** repeatedly to access the older generations of the same variables. This class is known as the **CONTROLLED** class of storage and represents a significant departure from the conventional methods, of storage allocation of other high level languages.

The fourth class of storage is known as the BASED class. Every based variable has a pointer associated with it called the pointer variable. By setting the pointer to different addresses in memory the value of the BASED variable can be set to the values of different variables. In other words, no memory storage need be allocated to a BASED variables, for it takes on the value of the variable currently pointed to by its pointer. It is mainly used in list processing and is described in Section 3.

2.5 Data Transmission:

PL/I provides a versatile repertoire of I/O instructions which to date have not been implemented in full in any of the machines built so far. This in itself speaks for the level of sophistication reached by the language as regards data transmission to and from the processor.

The I/O in PL/I is designed to interact closely with the supervisory software under which it functions and to keep track of input sources and output destinations.

Basically, there are two types of data transmission in PL/I - stream oriented and record oriented.

Stream Oriented I/O:

In this mode of transmission, data is considered to be available in a continuous stream of information with no conceptual breaks. Thus, when a command is issued to 'GET' a piece of data, then that piece of data is located from the input stream at a

point where the last command to input was terminated. Similarly when a command is issued to 'PUT' a piece of data on an output medium, then it will be written from where the last PUT command ceased to write. Notice the difference between Fortran READ/ WRITE statements and PL/I's GET/PUT command. In Fortran every READ/WRITE command begins a new record, effectively terminating the old record when the last READ/WRITE command ceased to read/write.

The second mode of transmission is known as the record mode of operation. In this mode, it is assumed that data is prearranged in chunks of information which is then transmitted to or from the processor. The transmission does not involve any prior conversion of data, but is literal, whereas in case of stream oriented I/O conversion will be performed.

It is evident that record oriented I/O will be faster since a) no conversion is performed and b) it explicitly recognises breaks in the incoming or outgoing data. Thus it is an ideal mode of operation when we are dealing with intermediate results which have to be stored temporarily on secondary storage medium. The stream oriented I/O has also its advantages as it allows I/O to be performed with or without editing and permits the novice to specify the actual names of the variables along with their values in the I/O data stream.

The File Concept:

Although the file concept is familiar to most ~~for~~Fortran users, it is not quite the same in PL/I. A file in PL/I has a symbolic name just as any other variable, (as opposed to Fortran files which are referred to by logical units) and can be broadly classified as Input, output or Update files. An input file may not be used as an output file and vice versa. An update file can be used both as an input and output file.

We shall conclude our remarks on PL/I I/O by stating PL/I file declarations do not carry any specification regarding symbolic or physical storage unit on which the file is to reside. They do not even specify the type of storage medium to be used as repository of the file. All this must be kept track of by the supervisory system. Thus in so far as I/O facilities are concerned a high degree of machine and configuration independence has been incorporated. But this has its own problems in implementation as we shall see later.

2.6 Asynchronous Operations in PL/I:

We shall now briefly touch upon the features in PL/I which provide for execution of a program as a set of asynchronous tasks. Provision is made for the following type of operations:

1. Creating and terminating a task.
2. Synchronising a task.

3. Testing whether a task is complete or not.
4. Assigning and changing priorities of a task.

The definitions of the term synchronous and Asynchronous operations are given in Appendix R . The reader is urged to clarify the meanings of these terms before reading further.

Creating a Task:

It is necessary for at least one task to exist when a PL/I program is started. This is called the major task. Tasks which are initiated by the major task are known as minor tasks. A task is initiated by specifying a TASK or EVENT and/or PRIORITY option with a call statement. The procedure, thus called, will be executed asynchronously with the calling program.

```
Example:  DECLARE B TASK;  
          .  
          .  
          .  
          CALL PROC (A1,A2,A3) TASK (B);  
          .  
          .  
          .
```

The execution of the calling procedure is known as the attaching task and the execution of the invoked program is known as the attached task.

Enquiring the status of a Task:

```
DECLARE E1 EVENT;  
CALL PROC (A1,A2,A3) TASK (B) EVENT (E1);
```

When procedure PROC is initiated E1 is set to 0 and when it is completed, it is set to 1. Thus the attaching a task can enquire about the status of the termination of Task B.

TASK SYNCHRONISATION:

This is achieved by means of the WAIT statement whose use is illustrated below

```
P : PROCEDURE MAINS,
    DECLARE DO TASK, (E1, E2) EVENT;
    .
    .
    CALL A TASK (DO), EVENT (E1);
    .
    .
    WAIT (E1);
    .
    .
    END P;
```

In the above statements, the main task procedure proceeds smoothly after initiating task 'DO' until it encounters the WAIT statement. Here further processing in the major task is suspended till task B is completed whence E1 becomes equal to 1.

Terminating a Task:

A task is terminated when the task encounters an EXIT or STOP or RETURN or END statement.

Priority in Asynchronous Operation: In many on line processing systems, situations may arise which compel a task to be suspended and which require the execution of another task.

Such situations are handled in PL/I by assigning priorities to tasks.

The priority for a task is set by the statement

```
PRIORITY (task name) = expression;
```

```
or CALL P(A, B) TASK (T) PRIORITY (exp);
```

The expression in either case is evaluated and converted to an integer which is then assigned to the TASK as the priority of the TASK.

2.7 Compile Time Statements:

In PL/I provision has been made to instruct the compiler to modify the source text before actually beginning the translation. Such instructions are known as Compile Time Statements as they are executed at compile time itself.

They are similar in structure to other PL/I statements and are distinguished by percentage symbol (%) before the statement beginning. The main set of instructions in this category are the DECLARE, GOTO, IF, DO and the PROCEDURE statements. These statements may be inserted anywhere in the program. Each of the statement is illustrated by an example:

Declare Statement:

```
%DECLARE A CHAR;
```

```
%A = 'ACHANGED' ;
```

This will cause every occurrence of A in the program to be replaced by 'ACHANGED' if this instruction is given before any usage of A in the program. Some times a single replacement may trigger off a chain of such replacements and hence repeated scannings will be made till no further replacements are possible. Variables taking part in Compile time statements must be compile time variables and any compile time expression can contain only integer constants.

Compile Time GOTO:

Sometimes a portion of the text may not contain any compile time executable statements. In that case, or in case the programmer deliberately does not want certain compile time statements to be executed, he may use the GOTO statement.

Example,

```

MAIN :PROCEDURE OPTIONS(MAIN);
      % S1   ;      % S2 ;
      % L   : IF A+B=C THEN % GOTO L1;
                      ELSE % GOTO L2;
      .
      .
      .
      % L1 : S3   ;
      .
      .
      .
      % L2 : S4   ;
      .
      .
      .
      END ;

```

The above example also shows the use of the compiler time IF statement. Depending on the boolean value of (A+B=C) either L1 or L2 is executed.

The Compile Time 'DO' Group: This is usually used to minimise execution time at the cost of some more work during compilation and a little more object code.

Example:

%DO I = 1 TO 10 ;	Modified Text
X(I) = X(I)+1;	X(1) = X(1)+1 ;
%END ;	X(2) = X(2)+1 ;
	:
	:
	X(10)= X(10)+1;

The DO index must be a compile time variable and the WHILE clause is not permitted.

The Compile Time Procedure:

When the preprocessor which handles compile time instructions, comes across a compile time procedure, it bypasses it entirely and resumes scanning after the end of the procedure. When a reference is made to that procedure in a compile time statement, then it will be executed. Compile Time procedures may be invoked by non compile time statements also.

Example:

```

MAIN: PROCEDURE OPTIONS (MAIN);
    %P:PROCEDURE (A,B,C) CHARACTER;
        RETURN (A '+' B '-' C);
    %END ;
    :

```

```

Y = D+P(X,Z,10) ;
.
.
.
END ;

```

The preprocessor will replace the relevant statement by

```
Y = D+A+B+-10;
```

3. PL/I and List Processing:

When PL/I was conceived of in late 1964, the originators had not included list processing facilities in the language. List processing facilities were added almost as an after thought sometime in 1965-66. The based class of storage, as mentioned earlier, has great relevance to list processing. In the discussion presented below, the reader is assumed to be familiar with list structures.

In lists, the underlying principle is to break the relationship between physical and logical arrangement of certain items of data. In other words, we do not want to be tied down by having contiguous chunks of storage space allotted to a set of data items. By removing this restriction, lists enable us to shift data around in a convenient manner. Whenever storage is allocated to a based variable by an ALLOCATE statement the allocation of such storage (which is different from previous generations) must be indicated by setting the corresponding pointer variable. For example,

```

DECLARE B1, B2 BASED (POINTR);
ALLOCATE B2 SET (POINTR);

```

The SET key word is used to update the value of the pointer to the current storage address of the variable B2. More rigid control over allocation of based variable may be exercised by specifying an area where allocation is to be made. This is done as follows,

```
DECLARE A AREA AUTOMATIC (FIXED (10,0),...FLOAT(8,5),
                           CHAR (30)...);
```

This declaration reserves enough space for A to hold as many items as are specified within parenthesis.

Now if we allocate a variable in this area by

```
ALLOCATE B2 IN (4) SET(POINTR);
```

this would cause allocation of B2 in A if there is enough space in A. If there is not enough space available, an error condition will be raised.

We shall illustrate the use of the above attributes in generating a unidirectional list by the following example,

```
/* THIS PROCEDURE IS USED TO APPEND AN ELEMENT POINTED TO
   BY 'EP' TO A LIST WHOSE NAME IS 'ELEMENT'*/
LIST: PROCEDURE (EP);
  DECLARE EP POINTR, 1 ELEMENT BASED (EP),
                2   P   POINTR,
                2   V   VALUE;
  P=NULL
  IF TAIL=NULL THEN HEAD, TAIL=EP;
                  ELSE P(TAIL), TAIL=EP;
  END LIST;
```

In the above example, the first time a call is made to LIST the HEAD and TAIL pointers are set to the first element. Subsequently the TAIL is moved to point to the latest element added while

HEAD is undisturbed. Also the pointer P is correspondingly updated to form a unidirectional list.

The NULL built in function when used for initialization creates an empty cell and returns the address of the cell as its value. When used in the IF statement, it is used to check whether TAIL is pointing to a null list.

Yet another attribute which is useful in list processing is the OFFSET attribute. This is primarily used in conjunction with a based variable in a based area.

For example,

```
DECLARE ROOM AREA BASED (P1),  
        RUSH OFFSET (ROOM),  
        VAR FIXED (2) BASED (P2);  
ALLOCATE VAR IN (ROOM) SET (P2);
```

OFFSET will now contain the amount of offset between the start of the area ROOM and the variable VAR.

OFFSET variables differ from POINTER variables in one important aspect and that is OFFSET variable always gives the relative displacement while POINTER gives the absolute m/c address of a based variable.

In conclusion we can say that in PL/I list processing can be done as elegantly as in any other language while retaining all the data manipulation capabilities of other high level languages.

CHAPTER II

OVERVIEW OF THE PL 7044 COMPILER

This Chapter discusses the choice of a subset of PL/I the overall organisation of the compiler, its limitations and some techniques used in the implementation.

1. The Choice of a Subset:

One of the first decisions that one has to take in a compiler writing project is the extent to which the implemented language is to be loyal to the original language specifications. It should be borne in mind that, more often than not, language specifications are made without considering the implementation hurdles that certain aspects of the language may pose on existing machines. We wish to point out that PL/I is not an exception in this respect, rather, as has been discussed earlier, PL/I is more machine independent than most other high level languages. Thus, one is left with the difficult task of pruning the original language, in order to inject a high degree of viability in the implementation of the language.

Our decisions, in formulating the subset of PL/I, hereinafter to be referred to as PL 7044, have been guided by the principle of maximising the number of user facilities in the subset. We set out by aiming for a full PL/I compiler. As we met hurdles, we tried to overcome them, and if we failed for one reason or other, we dropped that feature which caused the hurdle.

Thus our method of attack has been 'to cross the bridge when we come to it'. We must admit that this method may have cost us in terms of a slightly inefficient code, and a rather unwieldy organisation. But the reader will readily accept this line of approach if we set forth the constraints under which the implementation was carried out.

Firstly, the time available to us was limited. It would have been difficult for us to complete the formulation before we started the coding. It was necessary for us to carry on formulation and coding side by side so that we could be sure that the ideas we put down on paper could indeed be programmable on the IBM 7044 without exceeding the limitations of the computer like memory space, execution time etc. Secondly the lack of previous experience in a project of this type, slowed down the progress to some extent. It was difficult to foresee problems before we were actually confronted by them. Lastly, the lack of detailed literature on implementation of the language on various machines forced us to constantly change our strategy on an ad hoc basis. This is one of the reasons why global optimisation and index register optimisation were not attempted.

2. Organisation of the PL 7044:

2.1 Choice of Number of Passes:

Compilers are normally organised in a number of distinct passes or phases, each phase carrying out a well defined task. The choice of the passes is, then, crucial in determining the

overall efficiency of code generated and the speed of the compiler itself. These two factors are contradictory in nature and usually a compromise is struck depending on the objectives of the designers. Those in favour of lesser number of passes, often advance the argument that by 'doing as much as possible in a single pass' one reduces the book-keeping involved and avoids the problem of evolving suitable intermediate codes between passes. Also the fewer the number of passes, the lesser the quantum of I/O activity, in general, and to that extent the compilation is faster. Following this line of thought, a two pass compiler was contemplated for PL 7044. But very soon it was found to present insurmountable hurdles as outlined in the next section of PASS 1. It is worth while to mention here, some of the advantages that accrue from a multipass compiler. A rule of thumb suggests that the greater the number of passes the more efficient is the code likely to be. Also, sometimes larger number of passes may be the only feasible answer to implementation of certain high level languages like PL/I, COBOL etc. on machines having small fast access memory and a large not-very-slow secondary memory. For example, if one were to try and implement PL/I on an IBM 1620 m/c it would be fair to guess the number of passes at around 20 or more ! On the IBM 7044 it was finally decided to divide the compilation work into 3 passes, the descriptions and necessities of which follow.

2.2 Pass I:

The primary aim of the first pass is to resolve certain apparently ambivalent situations in the program text. To make the point clear let us consider the following examples,

Example 1	PROGRAM A	PROGRAM B
	.	.
	.	.
	.	.
 2 + 3.I 2+3. +I . .
	.	.
	.	.
	.	.

In PROGRAM A '2+3.I' denotes a complex constant, whereas in PROGRAM B we have a simple addition of a 2 real constants and a variable I. If we wish to resolve the uncertainty at lexical level, it would be necessary for the lexicon to look-ahead as many as 3 lexical units ahead every time it encounters a simple constant. This will slow down the otherwise fast lexicon.

Example 2:

```
DECLARE (REAL, INTEGER, COMPLEX) STATIC;
DECLARE (REAL, INTEGER, COMPLEX) = 0;
```

As PL 7044 does not have any reserved symbols but only certain key words which could be used as ordinary identifiers also, in the above example it is impossible to say anything about the nature of the statement till we encounter the '=' sign or the keyword STATIC. It would be illogical to argue that the symbol table will come to our rescue in such situations.

Even if the symbol 'DECLARE' were declared as a 3 dimensioned variable and is available as such in the symbol table, we cannot categorize the statements as an assignment or declaration statement merely by scanning the first symbol 'DECLARE'

Example 3:

PROG - 1	PROG-2
MAIN:PROCEDURE OPTIONS (MAIN);	MAIN:PROCEDURE OPTIONS (MAIN);
·	·
·	·
DECLARE B REAL ;	DECLARE B REAL ;
·	·
·	·
CALL SIN (A,B);	CALL SIN (A,B) ;
A:D=0 ;	END MAIN ;
END MAIN ;	SIN:PROCEDURE ;
SIN:PROCEDURE ;	·
·	·
·	·
·	·

In both the above programs, upto the statement 'CALL SIN(A,B)' there is no information about the symbol A, therefore no conclusions can be drawn as to its nature. In program-1 it turns out to be a label constant, while program-2 expects default attributes to be assigned to A. The problem becomes more acute because of the conversions that have to be performed when arguments and parameters in procedures do not having matching attributes. The problem becomes doubly compounded when we add a DECLARE statement in the main procedures of PROG-1 and PROG-2 as follows:

```
DECLARE SIN GENERIC (SIN1 WHEN (label, real),  
                     SIN2 WHEN (real, real));
```

The first program would expect a call to SIN1 be made, while the second would require a call to SIN2 be made.

Thus, in all these examples, the first pass can come to our rescue. In the first example it would have collected enough information about the nature of the constant, so that when second pass requires a lexical unit at that point in the program, there would be no difficulty in supplying it with a complex constant in one case and a real constant in the other. In the second example, the classification of statements carried out by the first pass helps in resolving the ambiguity for the 2nd pass. In third example the first pass would have collected all the labels of each block and made it available in a group to the 2nd pass with the result that at the beginning of each block the 2nd pass knows exactly which symbols constitute label constants and which of them have to be assigned default attributes. Certain other duties are also assigned to the 1st pass to make things easy in the 2nd pass. These are detailed in the Chapter on 'FIRST PASS PROCESSOR'. We have enough evidence in hand to justify the existence of this pass.

2.3 Pass - II:

In this pass, most of the syntax analysis and all of the semantic interpretation are carried out. But for the limiting factor of the memory space available, code generation could easily

have been included in this pass, rendering the 3rd pass redundant.

The syntactic analysis is a decentralized process and is carried out by several statement decoding routines. This strategy has a clear advantage of being fast compared to a centralized syntactic analysing scheme. The advantage accrues from the fact that each decoding routine not only checks the syntax but also assigns the semantic interpretation simultaneously. Such a possibility is precluded in the case of the centralized syntactic analysing scheme. A slight disadvantage of decentralization may be the larger core storage requirement, but, in any case, since we need a separate pass for code generation, it is not worth while to save a few location and lose in time.

The second pass has a set of symbol table management routines which are used by the decoders for getting/adding information from/to the symbol table. Since in PL/I language scalar expressions could be used at every conceivable point in the program, the expression handler is another facility which is utilized by the decoders. The expression handler has 3 inter-related routines - the buffer handler, arithmetic processor and the sorting routine. The motivation for this type of organization of the expression handler, and the tasks performed by the 3 routines have been detailed in Sec. 3.2 of this Chapter.

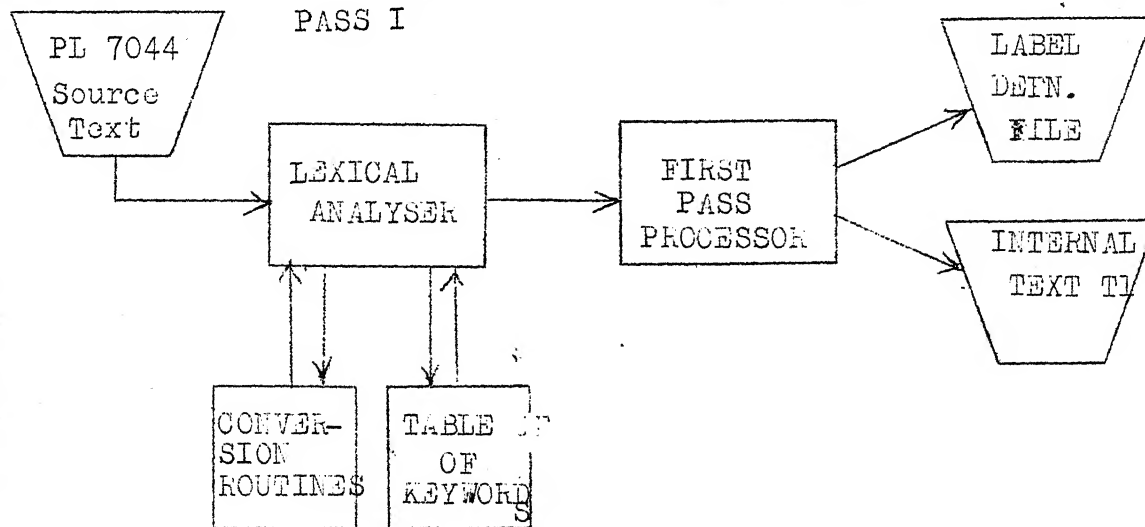
It is worthwhile to mention at this point, that even though the bulk of the code is generated by the 3rd pass, a small portion of it is generated in the 2nd pass also. Coding which is independent of the logic of the program falls under this category. Data lists, constants, file declarations etc. form a part of the code outputted during the second pass. This is in keeping with the basic philosophy of doing as much as possible in every pass.

2.4 Pass - III:

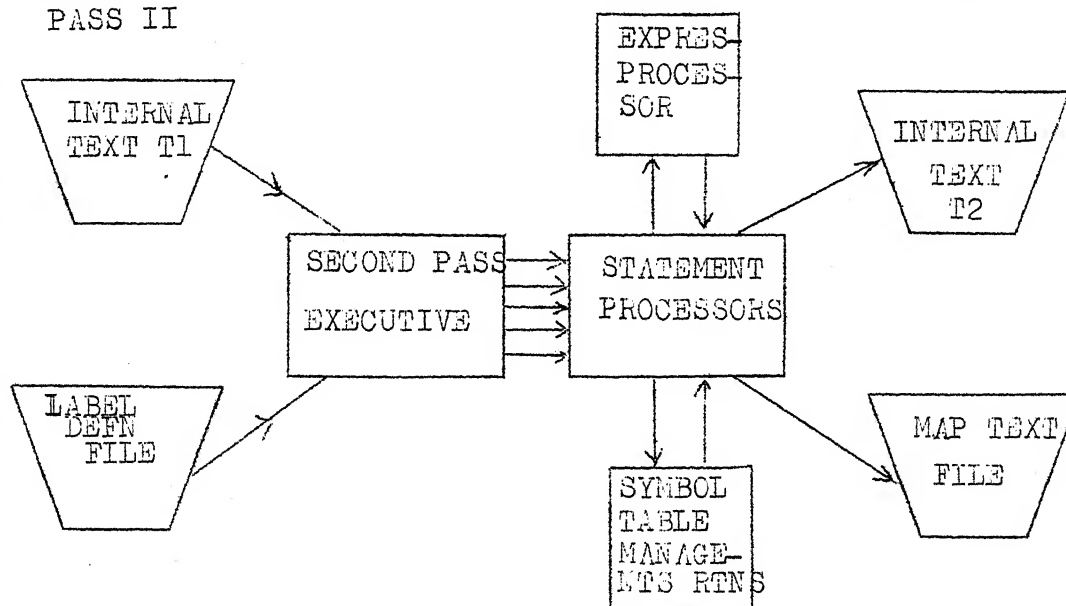
As has already been mentioned in earlier sections, the 3rd pass generates the final coding in a form acceptable to the MAP assembler. Apart from code generation, optimisation of temporaries is also carried out in this pass. No other complications are tackled at this stage and hence the 3rd pass constitutes, logically, the simplest pass of the three passes.

Four work units are required by the PL 7044 compiler. The output of the first pass is held on Work Unit 1. This output is read in by the executive of the second pass. The label definitions, block predecessor table and the DO table are outputted on Work Unit 2. In the second pass, the tables are initially read into pre-assigned locations, and at the beginning of each block, the label definitions collected by the 1st pass are entered into the symbol table. The intermediate code generated by PASS II is outputted on Work Unit 3 while the final MAP coding is made available on Work Unit 4. It is not possible to further optimise

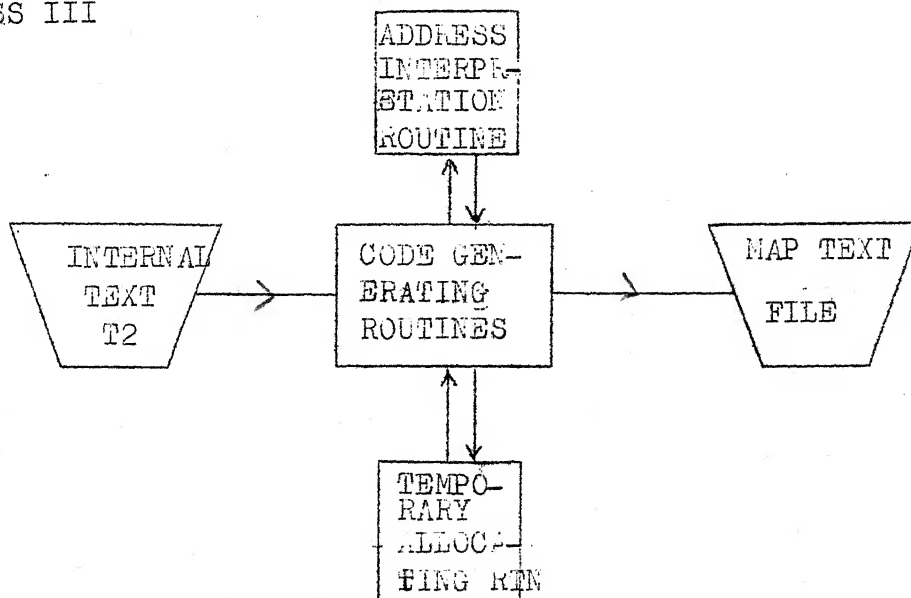
PASS I



PASS II



PASS III



on the number of Work units because during the 2nd pass, all the four work units are active.

The three passes may be edited on the System Library as three phases operating under the Processor Monitor IBJOB. The IBJOB will hand over control to the 1st pass via the System Loader and each of the passes will return control to the System Loader with a request to load the next phase.

3. Implementation Details:

3.1 Memory Management:

3.1.1 Storage Classes:

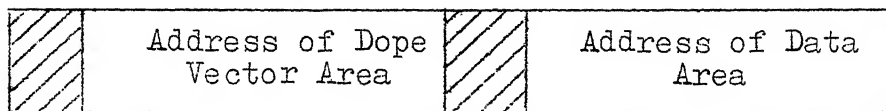
PL 7044 allows a programmer a choice of two storage classes - STATIC and AUTOMATIC. The other two classes of storage namely, CONTROLLED and BASED are not available because they can lead to fragmentation of memory in the absence of hardware facilities like segmentation and paging. The fragmentation will arise in a linear memory space, such as is available on the 7044, in the following manner. The CONTROLLED class of storage allows a programmer to allocate and free memory space for a particular variable in accordance with his requirements. He may, for example, create several generations of the same variable and then release storage for any or several of these generations. Now the order in which he creates successive generations need not be the exact reciprocal of the order in which releases them. In other words, in many programming situations the last-in-first-out principle may not be followed and this can lead to fragmentation of memory space.

PL 7044 compiler analyses storage requirement for the entire program and schedules the allocation of storage as it is needed. At run time, a general pool of storage cells is maintained which receives storage cells as when they are released and allocates them to other data items as and when they are demanded. Thus a data item may not be allotted the same memory space at two different allocations. The difference between the two classes of storage STATIC and AUTOMATIC is intimately connected with the partitioning of memory into blocks and will be discussed in Section 3.1.3.

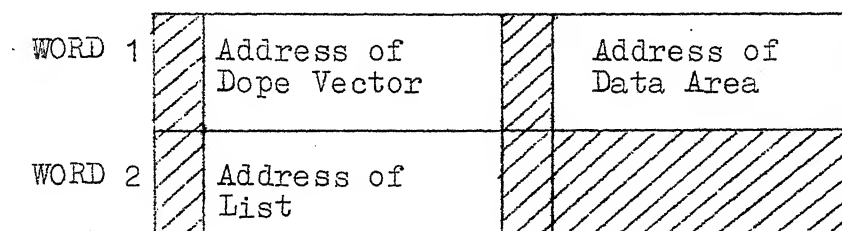
3.1.2 Representation of Various Data Items in Memory:

The rules for forming constants and their internal representation are given in Appendix B and will not be touched upon here, since they are self explanatory. Appendix D lists variable items under three headings - arithmetic variables, string variables and label variables. The internal representation of these types have also been given in detail and will not be discussed here. We shall concentrate our attention here to data aggregates and how they are handled during compilation and run time.

Internal Representation of Arrays: The array is represented in PL 7044 by three chunks of memory space called the header, the dope vector and the actual area containing data. The header area is a single 7044 word whose format is shown below.



This is the format when the array contains complex, floating point integer, character or bit string data items. In the case of an array of labels with a permitted list (See Appendix D for a description of label variables with a list) the header area consists of two words having the following format



The dope vector area contains information about lower bound, the range and the multiplying factor for each of the N dimensions of the array. It also stores an address which is useful for accessing an element of the array in the data area. Thus the space requirement for the dope vector of an N dimensioned array turns out to be $3*N+1$ locations.

Internal Representation of Structures:

The motivation in having structures and the facilities to manipulate them has been outlined in Chapter I. We shall concentrate our attention here to the internal representation of structure on IBM 7044.

A structure is represented in core storage as a linear collection of all base (or leaf) elements if the corresponding tree is scanned from left to right. Thus for example,

DECLARE 1 PAYROLL,

Level

2 NAME,

1.

2 HOURS,

2. NAME

3 REGULAR,

3 OVERTIME,

3.

2 RATES;

PAYROLL

HOURS

RATES

REGULAR

OVERTIME

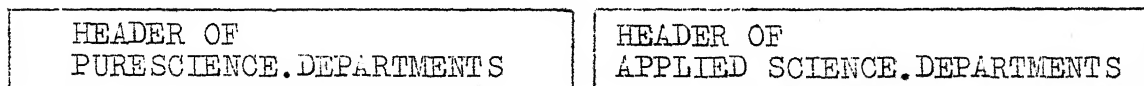
will be assigned storage as follows:



Each base element in this collection is treated as if it has been declared as a variable. In fact, at execution time, the structure loses its identity. Each base element is treated as a variable for addressing purposes. Of the two extended data aggregates - arrays of structures and structures of arrays, the former is not provided. Structures of arrays have been taken care of, however, and their internal representation is best illustrated by the following example:

DECLARE 1 IITKANPUR, 2 PURESCIENCE, 3 DEPARTMENTS (3),
2 APPLIEDSCIENCE, 3 DEPARTMENTS (7);

The internal representation of the above structure, which has two arrays as its base elements, is as follows



Thus, if a base element of a structure is an array then its header word gets inserted in the linear collection of base elements that we talked of earlier.

Representation of Formal Structure:

A major structure or a minor structure may be passed on as an argument in a procedure call. It is the responsibility of the programmer to make sure that the corresponding parameter of the called procedure is also a structure whose specifications (attributes) and hierarchy match those of the transmitted structure. When such is the case, the formal structure parameter (henceforth to be called formal structure) is once again represented by a linear collection of base elements. Each such base element is then treated as if it has been explicitly declared as a formal parameter. As in the previous case, the formal structure then loses its identity. There is, however, one important difference between normal and formal representation. Whereas in the case of normal structure, no storage cell is allotted to the major structure, in the case of the formal structure, the major structure is allotted one word for purposes detailed below.

When a formal structure (either a major or minor) is transmitted as an argument in a procedure reference, it is necessary to transmit the address of the first base element of the minor or major structure in the parent (or normal) collection of base elements. This facilitates uniform treatment of formal structures no matter how many times the same structure or a part of it is transmitted through a number of procedure references. Thus, it is this address of the first base element of the structure (major or minor) in the parent collection that occupies the

word allotted to the formal structure parameter. The above discussion is illustrated by the example below:

```

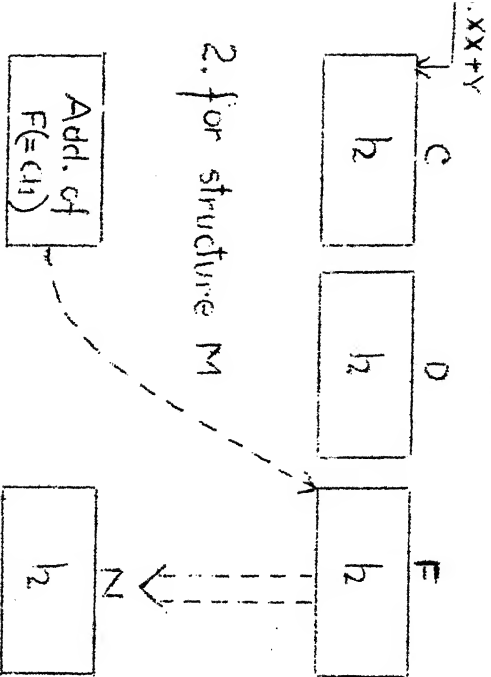
MAIN:PROCEDURE OPTIONS (MAIN);
  DECLARE 1 A, 2 B, 3 C CHARACTER (10), 3 D CHARACTER (5),
          2 E, 3 F (10,10) COMPLEX, 3 G, 4 I (10) LABEL,
          3 J, 4 K (10,10,10), 4 L BIT (10);
  .
  .
  .
  CALL PROC1(A.E);
  .
  .
  .
PROC1:PROCEDURE(M);
  DECLARE 1 M, 2 N(10,10) COMPLEX, 2 X, 3 P(10) LABEL,
          2 Q, 3 R(10,10,10), 3 S BIT (10);
  .
  .
  .
  CALL PROC2 (M.Q);
  .
  .
  .
PROC2:PROCEDURE(X);
  DECLARE 1 X, 2 Y (10,10,10) REAL, 2 X BIT(10);
  .
  .
  .

```

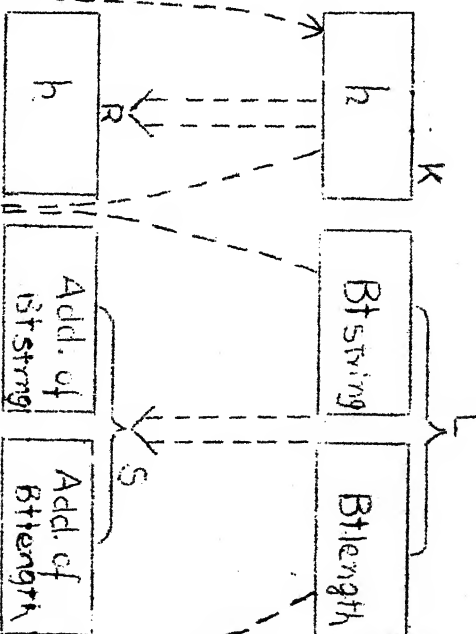
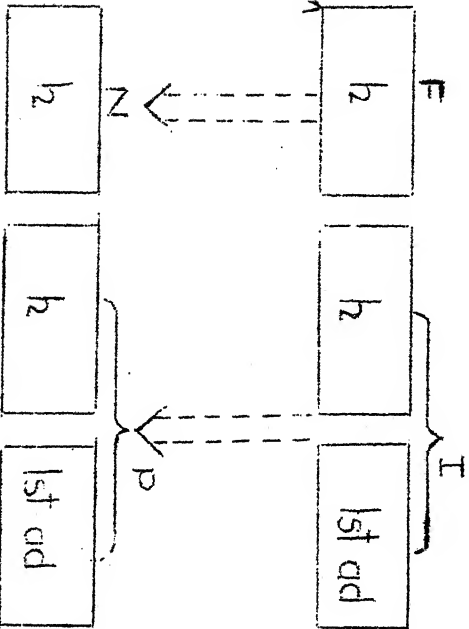
The storage representation of the three structured iis shown in Fig. OC-1.

Note that for formal structures M and N, the cells allotted to them point to the first base element under the respective major/minor structure of the parent structure. When A.E is transmitted from the MAIN procedure to PROC1 there is no difficulty in calculating the address of the base element of A.E, namely F. It is equal to $(BS.XX+Y)+(3-1) = a_1$ say) where BS.XX+Y is the starting address of the normal structure in core. When M.Q. is transmitted

1. for structure A



2. for structure M



3. for structure X

1st ad. \equiv 1st address
 $h \equiv$ header of an array
 $Bt \equiv$ Bt

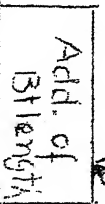
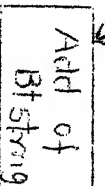
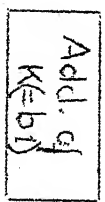


Figure 00-1

to procedure PROC2, the address of element K in the parent collection is arrived at by adding the offset of the first element of M.Q which is R to a_1 , i.e $b_1 = \text{addr. of K} = a_1 + (4-1) = (\text{BS.XX} + Y) + (3-1) + (4-1)$
 $= (\text{BS.XX} + Y) + 5.$

It is, worth while, at this point to comment on the splitting of the header word and the dope vector of an array into two distinct storage areas instead of merging them into one. If we had not resorted to splitting of the two sections, it will be necessary for us to transmit both these areas every time a structure of arrays is passed on as an argument in order to maintain the uniformity of treatment of all format structures. Thus an area equal to $3*N+1$ when N is the dimension of the array, will have to be copied every time, resulting in not only wastage of storage space but also in execution time because the copying is done at run time. Thus splitting the two areas avoids wastage of space and time and also allows the above scheme of internal representation of formal structure to function successfully.

3.2.3 Memory Partitioning:

In order to implement the block structure of PL/I, it was found necessary to partition memory software-wise. Before we proceed to detail the partitioning strategy, two definitions seem appropriate.

1. Block Representation: Each block of PL 7044 program is represented and referred to by an unsigned octal integer which is the number assigned to it in lexicographically ascending order. Thus, if the compiler has encountered $(N-1)_8$ blocks before encountering the current block, then the current block will in future be identified by N_8 .

2. Predecessor Block: A predecessor block is one that has, between its header statement and ending statement, the entire body of the block whose predecessor we are seeking. An immediate predecessor is a predecessor which is lexicographically closest to the block in question.

PL 7044 allows a maximum of 62 blocks in a program. All variables having the attribute 'STATIC' are considered to be belonging to an imaginary block '0' which is never closed and which encompasses the entire program and thus simulates the effect of a STATIC declaration.

A 'block predecessor table' is created during the 1st pass and is passed on to 2nd pass and finally to the run time ~~a~~ code. The Block Predecessor Table (BPT) carries the block number of the immediate predecessor of each block. Thus the BPT stores the lexicographic descendance relationship between blocks.

Available memory space is divided into 3 main parts,

- (1) Scalar storage area (herein after called as FIRST ORDER STORAGE)
- (2) Array Storage Area (herein after called SECOND ORDER STORAGE)
- (3) Save Storage Area (herein after called THIRD ORDER STORAGE).

The extent of the first category of storage is determined at compile time while the extent of the 2nd and 3rd order storage is determinable only at run time and these categories are allocated storage from the top and bottom of the free core space after the program code and first order storage have been loaded. More about the run time storage management is given in Section 3.2.4.

Scalar Storage Area (First Order):

An address of a scalar variable is an ordered pair, the first member of which is the block number N in which the variable is declared and the second member is an offset with respect to the starting of the storage scalar area of the block N. Scalar storage area may be divided under three heads, namely a) scalar element area, b) array dope vector area and c) temporary area.

Scalar Element Area:

All scalar elements defined in a particular block and the header words of arrays and character string variables are assigned storage in this area.

Array Dope Vector Area:

The representation of an array as described earlier consisted of three different areas. Of these, the dope vector is assigned storage in this area. The starting address of this area is the last location occupied by the scalar element area plus one.

Temporary Area:

The temporary storage requirements of a block constitute this area. The starting address of this area equals the last location of the dope vector area plus one.

Some Notations:

Starting Address of first order area of a block	BS.XX
Starting Address of Scalar Element Area (SEA)	BS.XX
Starting Address of Array Dope Vector Area (ADVA)	AS.XX
Starting Address of Temporary Area (TA)	TS.XX
A location in SEA may be referred to by	BS.XX+<offset>
A location in ADVA may be referred to by	AS.XX+<offset>
A location in TA may be referred to by	TS.XX+<offset>
Last first order storage location of a block block + 1 \equiv	BE.XX

In the above notations 'XX' represents the block number in octal and 'offset' is the position of the location relative to the start of each area.

For each block the following relationships hold,

BS.XX	EQU	BE.P(XX)
AS.XX	EQU	BS.XX+ \neq (SEA)
TS.XX	EQU	AS.XX+ \neq (ADVA)
BE.XX	EQU	TS.XX+ \neq (TA)

where $P(XX)$ = immediate predecessor block number.

The maximum number of first order storage is calculated as follows:

Step 1:

$$\#(0) = \text{SEAL}(0) + \text{DPVAL}(0) + \text{TA}(0).$$

Step 2:

At the end of each block the total first order area required till then is calculated as

$$\#(B_i) = \#(P(B_i)) + \text{SEAL}(B_i) + \text{DPVAL}(B_i) + \text{TA}(B_i)$$

where $P(B_i)$ is the immediate predecessor of the i -th block and SEAL, DPVAL and TA all have same notation as above,

Step 3:

$$\left. \begin{array}{l} \text{Maximum first order} \\ \text{Storage Required} \end{array} \right\} = \max_{i=0,1,\dots,N} \{ \#(B_i) \}$$

Let the maximum of first order storage be represented by MXFSAR.

At the end of the 3rd pass MXFSAR is available to the compiler, and it then generates the following code

```
BS.00 EQU *
      BSS <MXFSAR>
```

The 3rd pass also prepares a 'block storage table' and outputs it along with other code. This table consists of the starting address and ending address + 1 of the first order area of each block. It is used by the prologue routine during run time to effect saving and restoring of 1st order area in the 3rd order storage area.

The general format of an entry in the block storage area is,

PZE BE.XX,,BS.XX

where,

BE.XX and BS.XX have the same connotations as before.

Appendix F illustrates all that we have discussed about block structures by working out the actual codes that will be generated for the example in Appendix E.

Saving and Restoring First Order Storage:

As can be seen from the illustration in Appendix F, two or more procedures will share the same storage space if they have the same immediate predecessor. This is because the scalar storage area of one such block is inaccessible to the other, thus enabling the prologue routines to save, use and restore the same physical area over several procedures having the same immediate predecessor. Note that this technique will fail if the 'TASK' attribute were to be implemented because two procedures may be under execution at the same time and this will not permit such overlaps.

Let us now discuss, briefly, how the prologue routines calculate the amount of core storage (from first order area) to be saved in the SAVE area (third order area). At the time of invoking a procedure, the value of (BS.<called-procedure> - BE.<calling-procedure>) is calculated. If it turns out to be positive or zero then no saving need be done, since it implies that the block being opened does not share any first order area with the calling block.

On the other hand, if a negative value is obtained, the prologue routine saves an area equal to $|BS.<called> - BE.<calling>|$. This count and the position in the ^{procedure} 3rd order ^{procedure} area is carried along in the run time stack which is the topic of our discussion in the next section (3.2.4).

At the time of closing a block, and returning to the calling program any area that has been saved during invocation must be restored. This is achieved by consulting the linkage cells of the block in the run time stack (Section 3.2.4). We shall defer our discussion on the exact mechanism of restoring till the end of next section.

3.2.4 Run Time Stack Management:

The entire free core storage available after loading the program code, first order storage requirements, supporting sub-routines from the system library and the I/O buffer requirements is converted into a double ended stack which can be accessed from both ends. Fig. OC-2 shows schematically the runtime stack. The stack comprises the 2nd order storage area of each block described in Section 3.2.3, the 3rd order storage area and certain linkage cells which are vital for transfer of control from one block to another and other book keeping operations which will be outlined below.

The general format of the linkage cell is as follows:

	L.P.P.		return Addr.
	W.A.P.		# of words saved
	DOSRNO		Calling Block No.

Notations:

P.P.P. (Present Procedure Pointer) - This is a one word area which contains the address of the first of the 3 locations which constitute the linkage cell of the block which is being currently executed.

L.P.P. (Last Procedure Pointer) - This is the value of the P.P.P. when the invocation of the present block was made.

W.A.P. (Work Area Pointer) - It is the address of the first free location after arrays and character variables (Second order storage Area) have been allocated memory space. This denotes the address of the first free cell available for assigning temporaries requiring more than 2 words.

F.C.P. (Free Cell Pointer) - It is the address of the first free location after multiword temporaries have been assigned.

At the end of each assignment statement or expression using a multiword temporaries, the arithmetic processor generates code to bring the Free Cell pointer back to the value of the Work Area Pointer. This results in a slight inefficiency in the use of available space, since not all the temporaries will be required at the same time and some amount of sharing is possible. But the

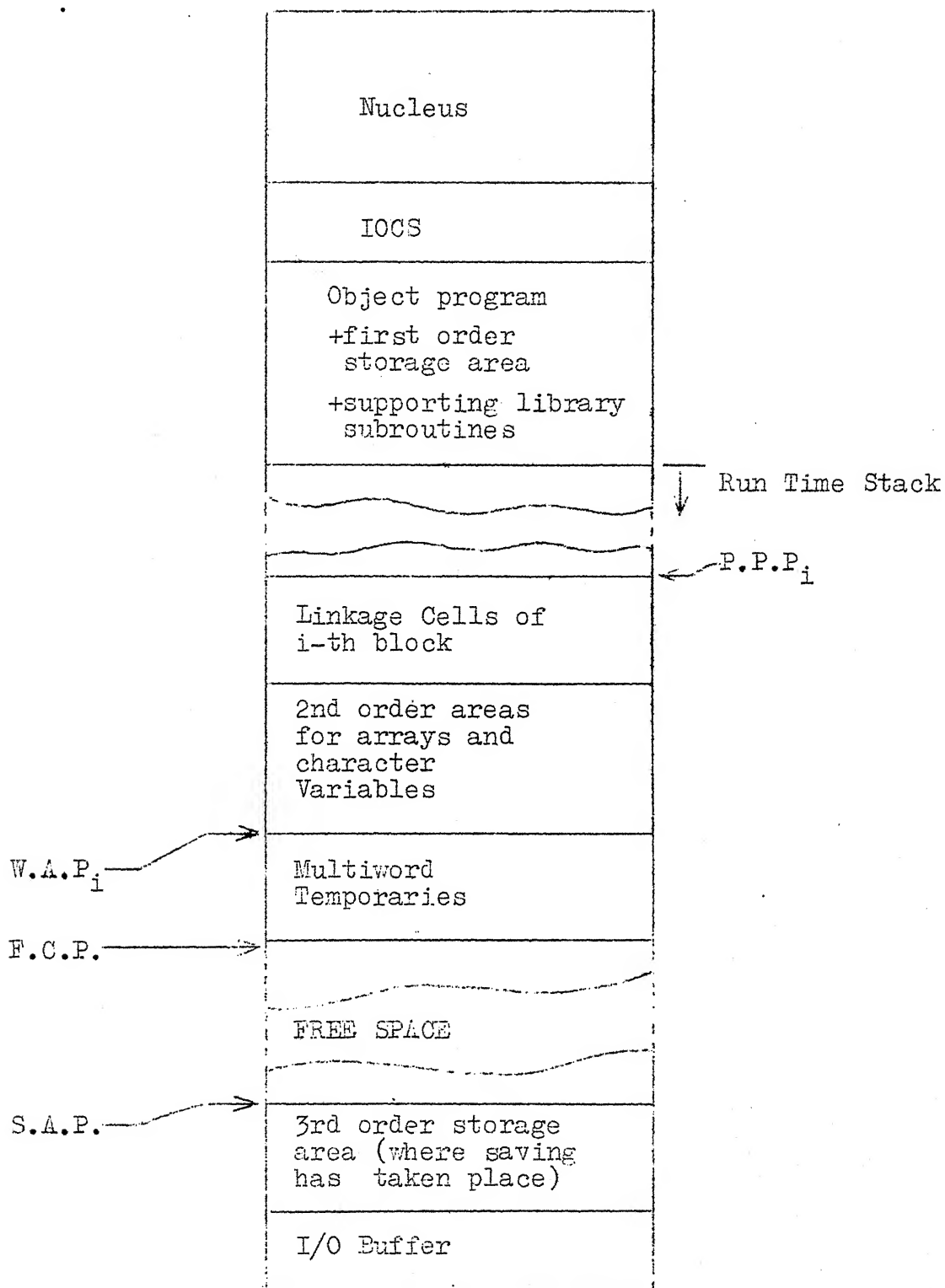


Fig. OC-2 : Memory Map at Execution Time Showing Run-Time Stack.

logical simplicity of the present scheme of allotting all the temporaries at one shot has been given more weightage in our considerations of the two methods. Also, if the temporaries are not allocated and released according to the last-in-first out principle, it would be difficult to keep track of the resulting fragmentation of memory space.

S.A.P. (Save Area Pointer) - It is the address of the location upto which the stack is filled from the high core side with 1st order storage saved during procedure invocations.

Mechanism of the Linkage Cell:

1. L.P.P.- At the time of closing a block we must reset the P.P.P. value to the address of the linkage cell of the block which invoked the current block. This is precisely the content of L.P.P.
2. Return Address: For a 'begin' block, this value is zero, since it cannot be called out-of-sequence; for a procedure it contains the address complement of the point of invocation. This is necessary for continuing execution after a called procedure has been deactivated by a RETURN or END statement.
3. W.A.P. - This pointer is to be saved in the linkage cell of the current block, every time a procedure invocation is made. This is because the called procedure itself will alter the value of W.A.P. and in order to retrieve the value of W.A.P. after call is over, we must save the current value of W.A.P. It is evident that while entering a 'begin' block such a precaution is not

necessary since a begin block cannot be invoked from out-of-line.

4. # of words saved: It stores the number of words saved, if any, at the time of invoking the present block. This is needed for determining if any restoration is to be carried out when closing the block.

5. DOSRNO: It is the serial number of a 'DO' group if the invocation has been made from inside a DO group. This is to ensure that the destination of a possible GOTO statement in the current procedure is a valid destination.

6. Calling Block No: With the help of this information and the number of words saved if any, the run time stack management routines carryout restoration of 1st order storage area saved.

3.2 Expression Processor:

As the name suggests, the expression processor is primarily concerned with the generation of object language codes for expressions in PL 7044. The code generation of the assignment statement is also done by the expression processor. The occurrence of an expression is permitted in practically every statement and at every conceivable point in the program. Each situation delimits the expression by a different pair of terminal symbols. Hence the expression processor must recognise the situation from where it is called and output code according to the needs of the situation. Thus, the factors one has to bear in

mind while designing an expression processor may be enumerated as under:

(i) Ambiguity regarding the use of an operator: In PL 7044, '=' is used for both assignment and relational comparison. The choice to be made between the two meanings, is highly context dependent. Similar is the case with delimiters like ',', '(', ')', etc. Hence the expression processor should be designed to distinguish between multiusages of operators and delimiters.

(ii) Recognition of the domain of the expression: As mentioned in the opening remarks, the expression processor must recognise different pairs of end markers for different calls. For example, in `DO A(I,J) = ... ;` the expression processor will be called into action after the 'DO' has been recognised by the 'DO' handling routine. In this case the input for E.P. is delimited by 'DO' on one side and '=' on the other. Whereas in the case of `A(I,J) = B+C;` the E.P's domain is the entire statement. The E.P. should not return control at the '=' symbol!

(iii) Recognition of Error Conditions: Besides some terminal errors like illegal operator-operand sequence, recognition of an error condition may also depend on the special requests of the calling routine. For instance, in the example in (ii) ';' is a valid occurrence in `A(I,J) = B+C;` but not so in `DO A(I,J);`

(iv) Distinction between Pseudo Variable Reference and Built-in function reference: In PL 7044, there are several built-in-function

names which may also be used as Pseudo variable names. The distinction is made again on the basis of context. For example, in `COMPLX(A,B) = A+B+COMPLX(C,D);`, the first reference `COMPLX(A,B)` is a Pseudo variable reference whereas `COMPLX(C,D)` is a built-in function reference.

(v) Recognition of scalar and aggregate expressions: For obvious reasons, processing of aggregate expression (i.e. expressions involving arrays and structures) need more work to be done than in case of scalar expressions. Now, if arrays and structures occur in the argument list of a procedure reference only, the expression will still remain scalar in nature. But if they occur in the argument list of some built-in functions, the whole expression can become aggregate in nature. For example, `'B+SIN(D)'`, where D was declared to be an array, will be considered to be of aggregate type, while `'E+P(D)'`, where P is a procedure by declaration, will be deemed to be of scalar type.

(vi) Analysis of the expression and generation of codes: In expressions, subexpressions are grouped together by means of parenthesis or by the well known rules of precedence (e.g. $a*b+c*d \equiv (a*b)+(c*d)$). Hence the expression processor must follow these rules in the course of its analysis, decomposition and generation of codes.

(vii) Meeting the requirements of different calling routines: The object code generated for the same expression in different situations, in general, will be different. This is because,

different situations require different end results. For example, the occurrence of $A+B$, where both A and B are floating point variables, as format explicitors will necessitate the conversion of the end result into integer form. The occurrence of the same expression in a data list of an I/O command requires no conversion.

(viii) ~~Assurance~~ of Standard Side Effects: The result of the expression $A+B+P(A)$, where P is a procedure by declaration and has the property of altering the value of A - such an effect is called the side effect - will be different in two different interpretations namely $(A+B)+P(A)$ and $A+(B+P(A))$. This may cause the same program to give different results in two different implementations. Thus, in order to avoid any implementation dependancy in handling expressions, certain norms have to be introduced for tackling such situations. The norm that is universally accepted is that all codes relating to procedure references must take precedence over other codes in an expression. This will necessitate a final sorting of the processor output.

(ix) Separating the repetitive and non repetitive codes in case of aggregate expressions: For object code efficiency one would like to calculate the non-repetitive part of an expression once and for all outside the repetitive part. But in general, the repetitive and non repetitive parts will be interleaved in an expression. For this reason also, sorting of the code before finally outputting the code, is desirable.

The factors (i) thru (vii) can be taken care of by a single scan of the input by a single routine. But this will make the logic very involved. It was therefore decided to have a two pass scheme in the present implementation. The first scan takes care of factors (i), (ii), (iii), (iv) and (v) and is executed by a routine called the BUFFER HANDLER. The second scan is carried out by the ACTUAL ARITHMETIC PROCESSOR and accounts for factors (vi) and (vii). As has already been mentioned factors (viii) and (ix) are taken care of by the sorting routine.

In the present organisation, each calling routine will identify itself and its peculiar requirements by entering the buffer handler at a preassigned entry point. The buffer handler lays out the expression in a form that is recognisable by the actual arithmetic processor. The buffer handler then returns control to the calling routine after setting up relevant flags to indicate the type of the expression in question. It should be borne in mind that the calling routine calls the handler anticipating an expression in the particular situation in hand. It may turn out that there is no expression involved and this status is what is conveyed by the buffer handler when it returns control to the calling routine. Proceeding a step further, the calling routine examines the flags set-up by the buffer handler, and if the existence of an expression is indicated, a call to the actual arithmetic routine is put through. The arithmetic routine analyses the expression laid out in the fixed length buffer by the

buffer handler, generates intermediate code, and calls the sorting routine. This in turns sorts the code according to predetermined priorities and returns control to the calling routine. This set up resulted in three distinct advantages (i) the modular structure of the organisation permitted parallel debugging of all the 3 routines and easier logic in coding, (ii) the calling routine was able to check the validity of the delimiter returned by the buffer handler, and thus the number of entry points were less than it would have been if each caller required a separate entry point- In other words, some of the entry points in the buffer handler could be used by more than one caller. This advantage accrued because the buffer handler returned control as soon it had done its job- and (iii) since both the buffer handler and arithmetic routine are called by the initial calling routine, certain amount of flexibility has resulted. The calling routine is in a better position to indicate its requirements to the actual arithmetic routine than the buffer handler. Thus the number of entry points in the arithmetic processor is also kept to the minimum.

As an illustration of all that has been discussed before, let us consider the following example:

```

PUT LIST ( ( ( (A+B) *C)**E DO I= 1,2,3) ) ;
          0 1 2 3 3' 2'          1' 0'

```


Here (¹ is used for bracketing the 'DO' whereas (² and (³ are used for bracketing of expressions. In order to analyse this situation the I/O analyser calls the buffer handler only after consuming the 4 left parentheses. The buffer handler returns at) ³ because of its inability to find a matching (². The I/O analyser supplies the matching left parenthesis and again calls the buffer handler which returns control at) ² for similar reasons. This continues till the buffer handler returns control after meeting the 'DO'. At this point the I/O analyser knows that there exists an expression consisting of 2 pairs of parenthesis and a 'DO' group. It can now call the arithmetic routine for analysis of the expression laid out by the buffer handler.

3.3 Input and Output:

3.3.1 System Limitations:

One of the most powerful features of PL/I is in the area of data transmission to and from the processor. As has been mentioned in Chapter I, this is also an area where maximum machine independence has been incorporated. The existing I/O software on the IBM 7044 is rather inadequate to handle the sophisticated requirement of PL/I I/O, the main reason being the I/O software is essentially designed to cater to the needs of FORTRAN like languages. Thus in deciding upon the features to be incorporated in PL 7044, one has to contend with limitations imposed by the system I/O software some of which are enunciated below:

- (i) Of the three levels of IOCS available to the user only the second level-IOOP can accept requests for accessing randomly located records. Thus one has to write one's own buffering system (IOBS) if one wishes to incorporate non-sequential file processing.
- (ii) The I/O routines which handle Fortran I/O statements are designed to handle records of data and are incapable of servicing stream oriented I/O commands.
- (iii) File handling capabilities are relatively primitive. This is probably because the IOCS was written primarily to cater to FORTRAN like languages. For example no provision for checking validity of file operations is made except on system input and output files.
- (iv) Unlike Fortran, where a user may use a limited number of file names only (i.e. logical units 0 thru 7), PL/I is not restrictive in this respect. The linkage between file names (which are arbitrary) and the physical devices (or symbolic devices) is usually a duty performed by IOCS. No such operations are undertaken by the present system.

On the hardware side, absence of certain features like paging, backward reading tapes, makes it difficult or rules out certain features like BACKWARDS etc. Under the circumstances, two solutions presented themselves for consideration. First, one could revamp the entire I/O Software or atleast a part of it in order to suit the language requirements and second, one could

write a number of supporting routines which might bridge the gap between what was required of an ideal I/O software package and what was available. The first solution, though better, in the long run, was too prohibitive in terms of man hours required, and one had to accept, perforce, the second solution although it suffered from a certain amount of clumsiness which is inevitable in any patchwork.

3.3.2 File Handling:

Files are declared just as any other variables by a DECLARE statement. They can also be declared by an OPEN statement. For a complete list of valid file attributes the reader is referred to Appendix G. At the time when the file is first used in an I/O statement, file declaration code is generated and outputted. Also the file is given a unique number depending on its lexicographic appearance in the program and thereafter the file is always referred to by this number. Since file names in PL/I can have a length more than 6 characters and since MAP language will not allow more than six characters, file declarations are made by forming a name such as 'PLF.XX' where XX denotes the file number. Thus a declaration in PL/I:

DECLARE MASTER FILE INPUT RECORD; will result in a MAP file declaration

```
PLF.00 FILE  U00,*,BLOCK=257,DOUBLE,LRL=256,TYPE3,ROF=1,
            ETC  REEL,EOF=REOFX.,ERR=RERRX.,EOR=REORX.
```

In the above example we notice that the file name 'MASTER' has been transformed into PLF.00 because it was the first file to have been declared.

Secondly we note that the linkage between the MAP file name and the symbolic device is also straight forward i.e. PLF.00 resides on U00. U00 is the abbreviation for utility unit having the symbolic name S.SU00. Notice that S.SU00 can be any device disk, mag tape or any other device available in the system. The user must, of course, know that the order in which he declares his files is the order in which the utility units are assigned starting from S.SU00. If he declares more files than the number of utility units available, file declaration generation is suspended and an error message is given. We are disallowing multifile units because the user, in any case, cannot find out which of his files reside on a particular unit. As long as he restricts the number of files to less than or equal to the number of units available he can know for sure which files are on which unit.

As we have mentioned earlier, IOCS does not check the validity of operations on files. The file control block generated by the IBLDR does not contain enough information about files to cater to this need. It was, therefore, decided to maintain a file status block for each file, the format of which is given below:

PFx1			ADDr1
five storage words			
PFx2			

PFX1 = PZE for stream print files
 = PON for stream input files
 = PTW for stream output files
 = MZE for record input files
 = MON for record output files
 = MTW for record update files

ADDR1= link address of last file status block used in
 an I/O statement

PFXA== PZE if this status block does not contain useful
 information
 = MZE if the status block contains useful information

The area shown under hatching is required to save certain information regarding status of the file. For example, if it is an input stream file, it is necessary to save the position (in words and characters) up to which the file has been processed. When the processing of a file is interrupted these pointers (word pointer and character pointer) are saved so that when processing is resumed on the same file the next character is made available. This is in contrast to FORTRAN file processing which is record oriented. In this mode of operation, every time the file processing is started, a new record is either read in or written out.

The link address Addr. 1 is necessary for flushing output buffers (if any of the files happen to be output files) at the end of the program. The exit monitor follows the chain so created and if an output file is indicated, and if some operation has been

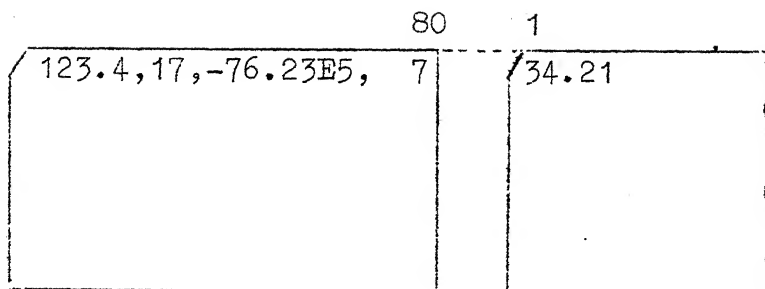
carried out on that file, than the corresponding buffers are truncated and written out. This ensures that no information is lost in the buffers.

3.3.3 Stream Oriented I/O:

The list directed, data directed and the edit directed I/O have all been implemented in PL 7044. The list directed I/O is essentially a free format facility intended to lighten the burden of I/O commands for the novice programmer. The data directed I/O is even more helpful to the newcomer in the sense that he can specify not only the values of the variables but also the names (consisting of valid identifiers) in the data stream. The edit directed I/O is meant for sophisticated editing of data items in accordance with format specification. Before we go on to describe the implementation of the three types of I/O commands it will be worthwhile to understand the mechanism by which record-oriented I/O routines of the system are made to appear as though they are stream oriented.

Let us specify, for simplicity, the card reader to be the input medium and the line printer to be our output medium. In case of FORTRAN, two successive read commands will cause two cards to be read, but in case of PL/I this is usually not the case.

Following figure shows two input data cards in which the values are punched in a free format (for a list directed input).



The first card is read in and the character and word pointers are initialized to point to the 1st column of the card. When a command to 'GET' a data item is issued, an input routine repeatedly calls a character fetching routine till it hits a non blank character. From this point onwards, the input routine will expect characters which are consistent with the definition of data constants.* When it encounters a comma or a blank it stops further scanning, does conversions if necessary for matching the target and source data items. Before quitting, this routine also updates the word and character pointer to point to the next character to be scanned. If the character pointer equals 80, it sets up a flag so the next time a command is given to 'GET' an item, the character fetching routine initiates a command to read a new card, resets the various pointers and then starts the card scanning. Similarly in the case of a line printer enough information is kept in order to detect the end-of-line situation and initiate commands for outputting the just completed line and request IOBS for buffer space for the next line.

* See Appendix B.

The mechanism of list directed I/O has been explained above in connection with the conversion of record oriented I/O to stream oriented I/O. We shall, therefore, proceed with a brief description of DATA and EDIT - directed I/O.

Data directed I/O requires that the names of the variables participating in the I/O command be available at run time. It should be borne in mind, that in no other situation in PL/I is required to furnish variable names at execution time. Thus, it is necessary for one to create a table of the names, attributes and the number and value of dimensions if any of the variable and maintain it at run time. Actually two such tables are created during the 2nd pass of the compilation. One is called the Compile Time Data Table whose capacity is limited to 500 words. The other table is the Run Time Data Table whose capacity is variable. Each entry in the Compile Time Data Table has the following .Format.

X	Y	Addr.1	
OFFSET	NDIM	AJTYP	BLKNO
			MNRTYP
DESCRIPTION OF 1ST DIMENSION			
DESCRIPTION OF 2ND DIMENSION			
DESCRIPTION OF 3RD DIMENSION			
DESCRIPTION OF 4TH DIMENSION			
DESCRIPTION OF 5TH DIMENSION			
		Addr.2	

FIG. OC-3

where,

X = no. of characters in the name

Y = no. of words in the name

ELKNO = block serial number where the name is declared

MAJTYP = 0 if it is a normal scalar variable

= 1 if it is a format Scalar variable

= immaterial if variable is array

NDIMN = 0 for a Scalar

≠ 0 for an array

Addr.1 = address complement of the last entry in the table

Addr.2 = address of corresponding entry in Run Time Table

(in the form of a label number)

The purpose of a Compile Time Table is to minimise the number of entries in the Run Time Table and thus save space during execution time. Every variable in a GET/PUT DATA statement is matched against the existing entries in the Compile Time Table. If a match occurs, no Run Time entry for that variable is created, and only the address of the existing run time table entry (Addr.2 in Fig.OC-3) is transmitted to 3rd pass. If a match does not occur, a Run Time table entry and a Compile Time Table entry are both created and appropriate addresses passed on. It is interesting to note that the Compile Time Table does not contain the actual characters comprising the name of the variable, but only the attributes of the variable. This is because of the fact that each variable during compilation is completely characterised by

their attributes alone, and there is no need to hold on to the symbols constituting the variable. In fact, one cannot say that symbols by themselves can uniquely define the variables, since the block structure of PL 7044 allows names to be non unique.

The format of the run time table entry is as follows:

DXXXXX

Word Count	Character Count		
VARIABLE SPACE FOR SYMBOLS FORMING THE DATA ITEM (MXM = 5 WORDS)			
		Address of 1st order storage	
		X	Y Z
VARIABLE SPACE FOR DIMENSION INFORMATION			

KXXXXX

where,

- X = No. of Dimensions
- Y = Major Type
- Z = Minor Type

In contrast to the Compile Time Table, the Run Time Table, holds the characters that form the symbol. This is as it should be, since in the input stream the only pertinent information available is the set of characters forming the name. Also, it should be noticed that whereas in the Compile Time Table each entry has a fixed number of words (=9), in the Run Time Table each entry has a variable format so as to optimise the number of locations required. Another difference is in the access mechanism of the two table. In the C.T.T. one accesses the entries (for searching) thru a linear chain, whereas R.T.T. is always examined selectively by the run time routines on the basis of the addresses provided to them by the translated code. In the figure depicting R.T.T. we have two labels per entry (if it is a dimensioned entry) or one label if it is a non dimensioned entry. The first label gives the address of the starting point of each entry while the second label gives the address of where the dimension information starts. Thus, for example, in,

```
GET DATA (A, B, C);
```

if A and B are 4-dimensioned variables and C is a scalar the following code is generated,

```
TSX  R,DATA,4
PZE  3
MZE  D00001,,K00001
MZE  D00002,,K00002
PZE  D00003
```

Thus, at execution time, enough information is made available for proper implementation of the GET/PUT DATA facility.

We shall now briefly touch upon the edit-directed I/O in PL 7044. Edit directed I/Os are always provided with format specifications. Format specification can either be given immediately after the closing of the data list, in which case they are called immediate format items or they may be given separately with a label prefix in which case they are called Remote Format specifications.

In so far as the implementation of the edit directed I/O instruction is concerned, there is not much difference between the two. What is of interest to us is the linkage between the format item and the data list. A cue has been taken from the manner in which Fortran Compiler does the linkage. The format is separately translated into a set of calls to various supporting routines as in Fortran, although the translation procedure itself is a lot more syntax and semantics ridden in case of PL/I formats than in the case of FORTRAN. Also it is quite likely that Format lists will be interleaved with arithmetic processor's coding in order to calculate expressions used as Format explicitors. The list of variables occurring in the data list is separately translated into a set of calls to a linkage routine which then sets up the connection between the data list item and the format list items. The basic strategy has been borrowed from the Fortran Compiler, but a variety of extensions have been incorporated.

3.3.4 Record-Oriented I/O:

The translation of record oriented I/O has been made relatively simple in the absence of the BASED class of variables for reasons mentioned earlier. No special techniques had to be used for the implementation of record oriented I/O other than the validity checking operations that need to be performed before executing Record-oriented I/O commands also.

3.4 Concluding Remarks:

In this chapter, we have tried to highlight, some of the techniques evolved to implement PL/I on IBM 7044.

The introductory nature of the chapter prevented us from going into too many details. However, if the interest of the reader has been sufficiently aroused to read further, the purpose of this chapter would have been amply fulfilled.

CHAPTER III

FIRST PASS PROCESSOR

1. The need for having a first pass processor has already been described in Chapter II Section 2.2. This chapter provides a more detailed description of the First Pass Processor (FPP).

The FPP has been entrusted with carrying out the following jobs.

- i) detection of complex constants
- ii) detection of '*' used as an operand
- iii) detection of 'iSUB' variables
- iv) detection of repetition factor before a string constant, and if one found, expansion of the string according to the repetition factor.
- v) collection of statement labels
- vi) (a) making a name table and supplying a unique name-key no. to each of the identifiers used in the program.
(b) supplying actual name string to second pass, if needed.
- vii) classification of statements
- viii) matching IF-THEN-ELSE in a compound statement and supplying ELSE if not used explicitly.

- ix) classification of END as to whether closing a DO, BEGIN or PROCEDURE.
- x) detection of multiple closure and supplying as many ENDS as necessary
- xi) making a do-predecessor-table and determination of total number of DO-groups used.
- xii) making a block-predecessor-table and determination of total number of blocks used.
- xiii) determining the place where all the declarations and specifications and (immediately after the heading statements) and signalling it to the second pass.

The input to the FPP is actual PL/I text and output of FPP is in an internal format. Internal representation of each lexical unit is given in Appendix A. During the first edition some validity checks are carried out on the source program, and if any error is detected, at the end of the 1st edition job is terminated and all further editions (2nd and 3rd) are deleted. The first edition uses two-work units for its output. All statement-label-constants are sorted out according to the block serial no. and are outputted on one work unit, where as code for actual statements is outputted on another work unit.

2. A short description of each of the activities listed, follows:

The first four activities are nothing but extension of the lexical activity, in general, since second pass receives

those four items as one lexical unit only, though first pass receives them as composed of one or more lexical units.

2.1 Detection of Complex Constants:

A complex constant can be in one of the two forms described below,

$$m_1 \pm m_2 I, \text{ or} \\ \pm m_2 I$$

where, m_1 and m_2 are either integers or floating point constants.

The first form represents a complex constant where as the second form represents only an imaginary constant (The FPP converts it into a complex constant whose real part is zero).

Both of these forms are then digested into one and given as a single lexical unit called complex-constant, whose representation is as follows:

7
real part
imaginary part

2.2 Detection of '*' Used as an Operand:

In PL 7044 language '*' is not only used as an operator but also as an operand, e.g. in the case of array cross-sections, star is used as an operand.

$A(1,*)$ represents a line $x = 1$ in a two dimensional plane. Consider $A(I*J,*)$. The first use of star is as an operator where as the second use is as an operand. The representation of '*' in the two contexts is as follows

'*' as an operator

-	10		4
---	----	--	---

'*' as an operand

			8
--	--	--	---

2.3 Detection of 'iSUB' Variables:

This lexical construction is used while defining the relationship between a base array and the defined array (i being the dimensional position). For example the statement `DECLARE A(10,15),B(15,10)DEFINED A(2 SUB,1 SUB);` defines two matrices. A is a matrix of 10x15 order, where as B is defined to be a matrix which is the transpose of matrix A. Since lexical analyser will break iSUB into two lexical units and would give i-as an integer and 'SUB' as PRW, the job of the FPP is to merge the two into one if there are no intervening blanks in between the two and it has been used inside parentheses. The representation of iSUB, then, is as follows:

+	i		9	(i = 1,2,3,4,5).
---	---	--	---	------------------

2.4 Detection and Application of Repetition Factors:

PL/I provides a concise and short representation of strings containing repetitive sub-strings e.g. if one has to define a bit string of length 30, containing all 1's, then writing thirty, 1's one after another would be quite a waste of

time. Instead it can be written as (30)'1'B, which is a much more elegant and concise representation. Similarly a character string 'ABCABCABC' can be written as (3) 'ABC'. The job of FPP is to detect its presence and then to expand the following string. The second pass is transmitted the expanded string, as if repetition factor was not used at all.

The activities described above were nothing but extensions of lexical activity, however the activities to be described now involve logical decision making and have nothing to do with lexical activity as such.

2.5 Collection of Statement Labels:

As described in section 2.2 of Chapter II, the most important task of FPP is the collection of statement labels. Statement labels can mean four different things at four different places in a program.

- (a) statement label prefixed to a PROCEDURE statement defines a primary entry point.
- (b) statement label prefixed to an ENTRY statement defines a secondary entry point.
- (c) statement label prefixed to a DECLARE statement refers to the next executable statement.
- (d) statement label prefixed to any other statement defines a statement label constant.

Apart from the classification given above, special mention should be made of the statement labels prefixed to BEGIN and DO

statements, since they are needed in determining the closure of groups and blocks by the END statement, if a label name follows the END statement.

Along with all this, FPP also determines whether label initialisation is needed and separates it from the label definition, e.g. consider a trivial PL 7044 program.

```

MAIN:PROCEDURE OPTIONS (MAIN);
    DECLARE A(10) LABEL;
    GO TO A(1);
A(1):PUTLIST ('LABEL INITIALISATION DONE');
    STOP;END;

```

In this case execution of program should print out the message LABEL INITIALISATION DONE, since A(1) would have been initialised to a statement label referring to the PUT LIST statement.

A complete description of code generated for label initialisation can be found under Section 3.5.3, Chapter DC.

FPP prepares a chained list of all statement label definitions. Each label is represented by two words. The compiler maintains a label counter which is used for mapping statement-label names to actual labels, generated in code. Each statement label field is given a unique label number so that it can be uniquely identified. All the labels prefixed to a statement get the same label number which ensures that they refer to the same location in the code.

The format of the label entries is as follows:

	DO serial no.	3	0	5	Blk no.	Statement-label- constant
	name-key-no.		label serial n number			

1 for integer, 0 for real

		blk no. code	1/2	Blk no.	<u>Entry constants</u>
	name-key no.		label serial no.		

Chr.-4:

1 for integer, 0 for real

1. for primary entry points (with PROCEDURE statement).
2. for secondary entry points (with ENTRY statement)

Blk no. is the block number in which the statement label is used. Name-key-number is the no. with which the name used as label is identified in the compiler. Each label is also given a DO-serial no., which is the serial no. of the do group open at the place where statement label is used. If no DO is open at that point it is zero. This no. is used to determine the validity of a GOTO destination since transfer to inside a DO group from outside is illegal.

Block to which code belongs and block to which statement label belongs are two different things. The distinction should be clearly understood. The statement label belongs to the predecessor block of the block to which the code belongs. The knowledge of the block to which the code belongs is needed for generating the Procedure Header Word(PHW)(Section 4.2, Chapter PC).

In the label chain the list of statement labels prefixed to a PROCEDURE/ENTRY statement is terminated by a special marker which has one of the four values 0-3. The interpretation of these values is as follows:

- 0 - entry has arguments but no RETURNS specification
- 1 - entry neither has argument nor RETURNS specification
- 2 - entry has arguments as well as RETURNS specification
- 3 - entry has no arguments but RETURNS specification is there.

This information is needed for generating the PHW as well as to analyse RETURNS if it follows. The information about the type of result returned by a function reference is kept in symbol table which is then used by arithmetic processor for generating proper code, including the code for conversion if necessary. In absence of any RETURNS specification default action is taken depending on the first character (conveyed to second pass) of an entry name. All the lexical units for RETURNS specification follow this marker word and RETURNS is itself ended by another marker word (all zero). Attribute analyses routine (Sect.3., Chapter DC) is called in at second pass to analyse the RETURNS specifications.

The label serial no. which is stored in the second word refers to the label number which is substituted in lieu of the statement label name. The label counter is increased by 1 for ordinary labels, and by 3 for entry names. The reason for increasing it by 3 for entry names is as follows.

If L is the value stored in the second word, L refers to the address where the code starts, L+1 refers to the address of prologue list (Section 4.2.1, Chapter PC) and L+2 refers to the address of the PHW which is stored in the symbol table.

2.5.1 Need for Chained List:

Because of the nested structure of blocks it becomes necessary to maintain a chained-list of all the statement labels defined. At the end of the first pass this chain is followed to sort out all the label definitions according to the block serial no. (The second pass processor at the beginning of a block loads all the label definitions for that block in the symbol table). For this chaining a table called 'label table' is maintained by the FPP, which has one word per block and stores the starting address where the list starts for that particular block and the end address where the list ends for that block. The list is made in the free core area starting at the lower address.

2.5.2 Actions to be Taken at the Time of Opening a New Block:

It is assumed that there exists an imaginary block which encompasses all the external procedures, and in the beginning, block '0' is considered to be open. Hence a label table entry corresponding to block '0' would be made to point to the first free cell in core.

Now when another block is to be opened, the block which was open till now is to be temporarily deactivated till the matching END is found for the new block which would reactivate the deactivated procedure. For this an extra word is added at the end of the list and its address is put in the address portion of the corresponding label table entry. The word whose address was till now there in the address position, also gets the address of the added word, thus making a forward linkage. Now an additional cell is added at the end of the list and its address is stored both in the address and the decrement position of the entry corresponding to the block to be opened, in the label table.

2.5.3 Actions to be Taken at the Time of Closing a Block:

While terminating a block, its predecessor block would have to be reactivated. The closing of a block involves adding a word containing zero, (erving as the end of the list marker for this block) and putting the address of this added word into the word whose address was stored in the address portion of the corresponding entry in the label table, thus making another forward link. For reactivating the predecessor block, one more word is added and its address is then put in the word whose address was stored in the address portion of corresponding label table entry and in the address portion of the label entry itself. This completes making the necessary forward links.

2.5.4 Special Treatment for Entry Statement:

The entry statement deserves a special treatment because though the statement label prefixed to it occurs inside a procedure block, yet it logically belongs to its predecessor block which was open before this block. This is in keeping with the treatment given to the primary entry points i.e., statement labels prefixed to the PROCEDURE-statement. Thus the whole process of deactivating the present block, opening the predecessor block, entering labels, closing the predecessor block and once again opening the present block is to be followed. To bring out the point consider the following example:

```

      MAIN:PROCEDURE OPTIONS (MAIN);
L1:L2:L3:L4:S-1;
AP1:IP2:AP3:PROCEDURE (A,B);
      DO-1;
      L5:S-2;
AP1:IE2:ENTRY RETURNS (CHARACTER);
      L6:END IP2;
      L7:S-3;
A)4:PROCEDURE (A) RETURNS (REAL);
      L8:S-4
      B1:BEGIN;
      .
      .
      .
      END B1;
L10:L9:DO-2;
      S-5;
      END MAIN;

```

The label table for the example given would be as follows:

START ADDR.	END ADDR.	Block <u>No.</u>	Name <u> </u>	Type <u> </u>
0	75	0	imaginary	Procedure
5	73	1	MAIN	Procedure
22	43	2	AP1, IP2, AP3	Procedure
58	71	3	AP4	Procedure
64	65	4	B1	Begin

Various linkages in the label chain are shown in the Figure FP-1.

Notation:

1. All cells marked '////' are control cells which are added to follow the linkages.
2. All cells marked '\\\\' (3, 20, 31 and 49) represent the type of entry names (presence/absence of arguments and RETURNS).
3. All cells with 'RE' (38 and 56) are the end of RETURNS markers.

At the end of the first pass the label out putting routine starts from the zero-th block, takes the starting address from decrement, follows the chain and at the same time outputs the chunks of words till the control word containing 0 (end marker) is reached. Same process is then repeated for other blocks also. This process automatically sorts the labels according to the block no.

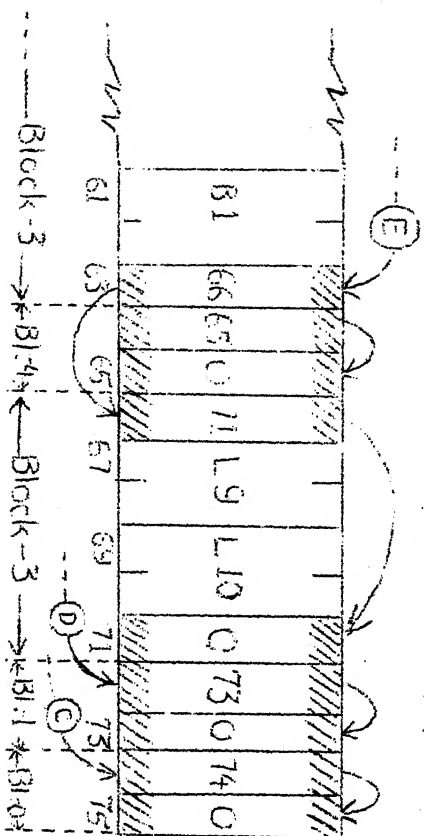
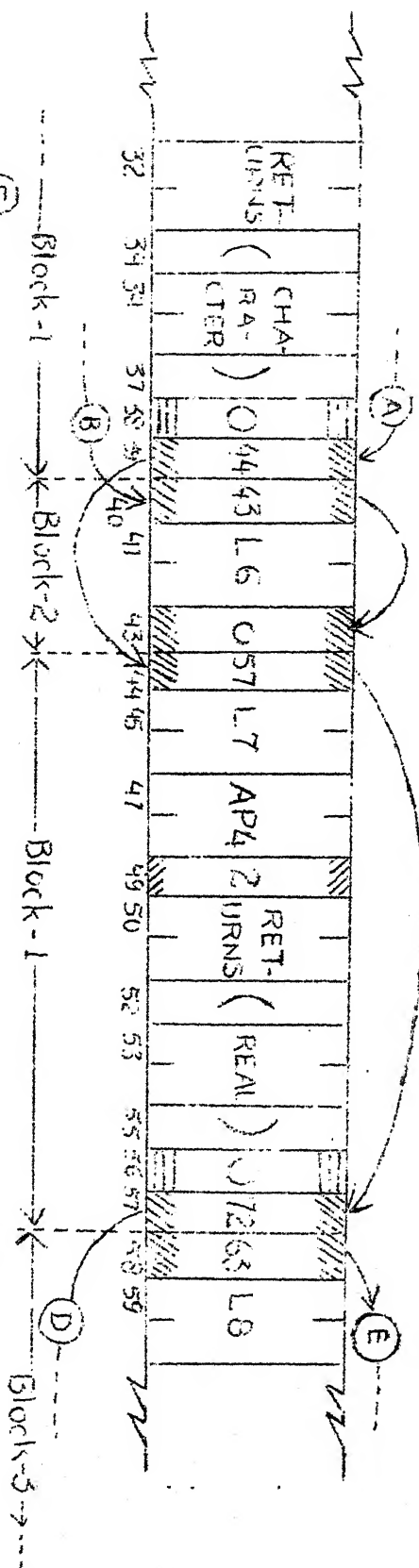
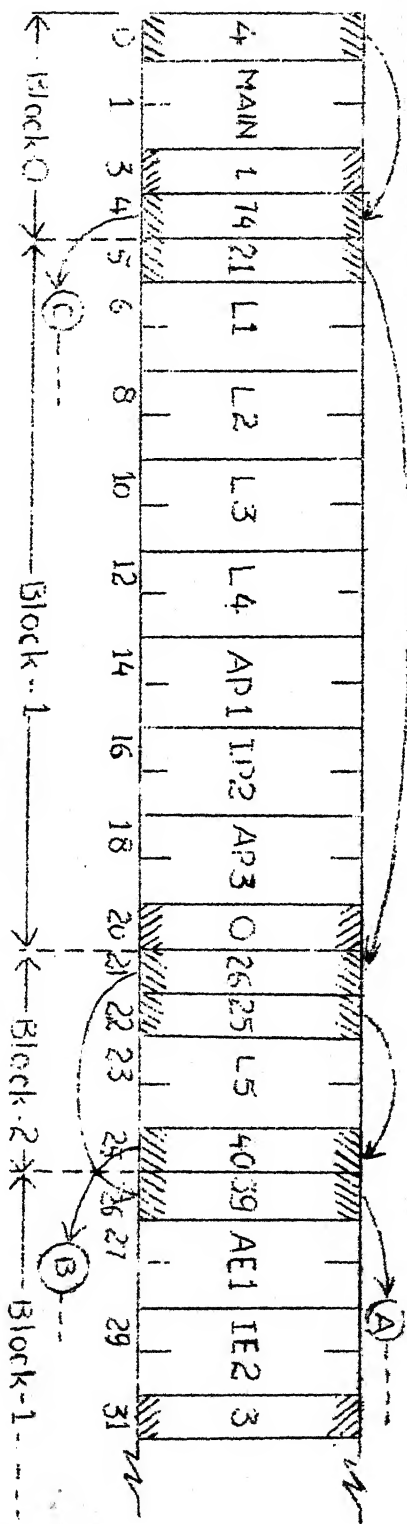


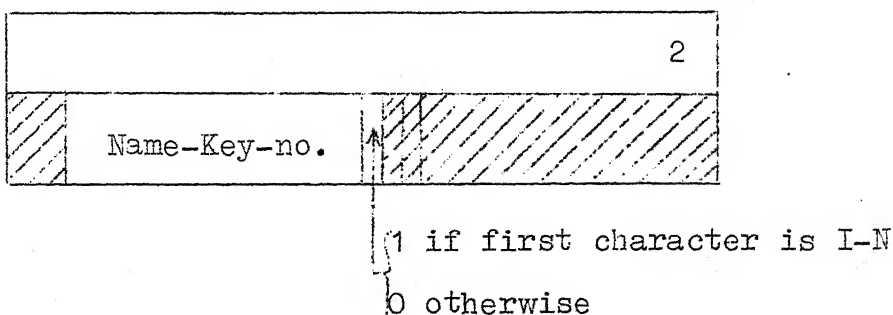
Figure FP-1

An interesting case is when there is no label defined in a chunk. The two control words come next to each other in such a case. In case of multiple closure this process for 'END' statement is followed for each of the blocks which get closed because of the multiple closure, as if an END statement was explicitly provided. This results in a cluster of control words as is evident at the end of the chain in Fig. FP-1.

2.6 Making a Name Table:

The actual name string for a symbol or identifier exists only in first pass. Second pass does not even get a feel of actual name string except in cases where it is absolutely necessary to convey the actual name string to the second pass. The name table is organised as a HASH TABLE of dictionary type. There is a table of size 1024 which stores pointers to actual name strings which are stored in words taken from the high core side.

Representation of an identifier, that is transmitted to the second pass is as follows



used in GET/PUT DATA statement, since first pass does not analyse the declare statements and as such it is unaware of elements contained in a major/minor structure.

One extra feature of this compiler is that it allows the programmer to use procedures written in 'MAP' by making a declaration DECLARE name MAP;

In such a case also, SPP would require the actual name string of 'name' so that an EXTERN card could be generated. This restricts the length of 'name' to less than or equal to six characters.

2.7 Before discussing further about other activities of FPP it would be appropriate to talk about the stack mechanism incorporated in FPP, which is needed for making logical decisions about multiple closure, matching IF-THEN-ELSE etc. The stack is constantly being updated throughout the FPP. The following statements either add/delete cells to/from the stack

```
DO
BEGIN
PROCEDURE
IF
ELSE
```

Deletion is done by other statements also, only if the top of the stack is of ELSE type.

Format of the Stack Cell:

Type of the cell	Type serial number		Last in Stack
+/-	Shielded If number		Last in stack of same type
	L1		L2
	L3		L4
	L5		L6
	L7		L8

<u>Type</u>	<u>Description</u>
0	DO
1	PROCEDURE
2	BEGIN
3	IF
4	THEN
5	ELSE

L1-L8 are the name-key-nos. of the statement labels prefixed to the DO/BEGIN/PROCEDURE statement, any of which can be used in the END statements. This puts a limit of 8 on the no. of labels that can be prefixed to either of DO or BEGIN or PROCEDURE statements. In the absence of labels prefixed to these statements, all these fields would be containing zeros. If only $n(n < 8)$ labels are used then the remaining $(8-n)$ fields would be containing zeros. Type serial no. is the serial no.

of that type of statement e.g. for a DO type cell it will be DOSRNO, and for a procedure/begin type cell it will be block serial no.

The address of the top most cell of the stack is stored in the location 'LSTANY' so that the stack could be traversed back words. The address of the top most cell of do type and block type, cells are also stored in the locations LASTDO and LSTBLK respectively. These are needed for making various decisions.

2.8 Classification of Statements:

FPF maintains a buffer where it goes on storing the lexical units constituting a statement. Some of the statements can be recognised immediately by looking at the next lexical unit e.g. a DO statement since either an identifier or ';' is to come after DO, if it is to be a DO statement. Care is taken here to see that PL 7044, does not have any reserved word hence DO:DO; is a valid construction where first DO is used as a statement label, and second 'DO' is used as a key word representing a 'DO' statement. This can go to any extent, some interesting examples follow.

- (1) DECLARE IF LABEL;
THEN: IF THEN \neq IF THEN IF = THEN;
- (2) RETURN : RETURN (RETURN);
- (3) DECLARE THEN LABEL;
IF: IF \neq THEN THEN THEN = IF;

PL 7044 has certain statements which cannot be recognised by a fixed number of look-aheads. In such cases a decision is taken only at the end of the statement one; such example is as follows:

```
DECLARE (A,B,C,D,E) REAL;
DECLARE (A,B,C,D,E)=REAL;
```

First statement is a 'DECLARE' statement where as the second statement is an assignment statement. All those statements where a left parenthesis can occur after the key words (viz. FORMAT, RETURN, DECLARE, ENTRY, PROCEDURE, IF, ELSE, DISPLAY etc.) fall into this category.

In such cases a note is made about the first keyword and at the end of the statement presence of an = sign (outside parenthesis) is determined to see whether it is an assignment statement or some other statement. Only this test is not sufficient in all the cases, then some other deterministic tests are made to ensure that the type of a statement is correctly determined (e.g. IF(A+B)*C = 0 THEN ...; ' though this statement has an = sign outside the parenthesis yet it is not an assignment statement.)

The FPP output for a statement is of following type

77777	← header
Statement type	
Statement number	
Actual Code	

Statement no. is conveyed to second pass so that in case an error is detected in second pass, it can be properly referenced to the program listing generated in first pass. However all the statements are not outputted in this form viz. END, BEGIN, zero argument entry etc. These are outputted as follows:

77777	
Identifying Information	Statement code

(The identifying information is block number, do number etc.)

The first word is a marker word which demarcates two statements from each other. This is used to check the correct transmission of code from first pass to second pass. At the end of a statement next word on tape should be this maker word only.

2.9 Matching IF-THEN-ELSE:

'IF-statement' is one of the most powerful logical mechanisms provided in PL/7044. IF statement is a compound statement, because it contains other program elements including do-groups and begin-blocks. It is this nesting property which imparts a strong decision making capability to the IF-statement. The nesting of program elements and IF-statements can continue to any desired level.

2.9.1 General Form of IF statement is as follows:

IF scalar expression THEN-Unit-1 [ELSE Unit-2]

Syntax Rules:SR-1:

Each unit is a DO-group, a begin block or any statement other than END, ENTRY, DECLARE, FORMAT or PROCEDURE statement. The unit may have its own labels.

SR-2:

The IF statement is not itself terminated by a semicolon. IF statement has two forms:

- (i) IF scalar expression THEN Unit-1.
- (ii) IF scalar expression THEN Unit-1 ELSE Unit-2.

The job of FPP is to supply a Dummy ELSE for every IF. Statement of Type-1.

Since either of the units can be an IF statement, this job of matching ELSE and IF has to be done with the help of a stack. FPP on meeting an IF statement adds two cells to the top of the stack, (one of IF type followed by a second cell of THEN type). The IF statement is broken into two parts for analysis, first part consists of 'IF scalar expression THEN' and the second part consists of 'Unit-1'. After analysing the first part, Unit-1 is given to FPP as if a fresh statement is starting. FPP generates three labels for every IF statement analysed.

1) Success Label: It is generated before the code for 'Unit-1' where control is to go if resulting bit string contains at least one-'1'. (The scalar expression is evaluated and converted to a bit string, if necessary).

- 2) Failure Label: It is generated before the code for 'Unit-2' where control is to go if resulting bit string contains all zeros.
- 3) End Label: It is generated at the end of code for 'Unit-2' where control would come after Unit-1 is executed in case of success.

For the IF statement of Type-1, the end label and failure label refer to same location.

All the statements(including IF but excluding those listed under SR-1) pop the top most cell of the stack if it happens to be of THEN type and generate a success label. The popping off of the 'THEN' type cell results in the exposure of IF type cell, which can be popped by an ELSE statement only. This scheme automatically matches the innermost ELSE and IF statement. For all the statements listed in SR-2, this situation constitutes an error.

As in the case of IF-statement, ELSE statement is broken into two parts - 'ELSE' and 'Unit-2'. Unit-2, like Unit-1, is resubmitted to the FPP for analysis. When an ELSE statement is met and the top of the stack is not of IF type, it indicates either an extra ELSE or a misplaced ELSE and constitutes an error. If top of the statement happens to be an IF type cell it is converted to an ELSE type cell and failure label is generated.

Now all those statements which have been given the power of popping the THEN type cell, have also been given the power of popping ELSE type cell, if the top of the stack happens to be an ELSE type cell. For statements listed under SR - 1, this situation also constitutes an error. For all the statements (except DO and BEGIN) which pop off an ELSE type cell, an end label is generated at the end of that statement.

Though DO and BEGIN statements also pop off the ELSE type cell, (they carry this information along with them) yet they do not force the generation of end label. This is done to shield the ELSE till a matching END statement is found and then the end label is generated. This ensures that the DO-group or BEGIN-block is treated as a 'Unit'.

If any statement other than ELSE finds the top cell of the stack to be of the type 'IF', it indicates that a dummy ELSE is to generated to match the IF. In such a case the IF-type cell is popped off, end-label and failure-label are generated at this point and this process is repeated as long as the top of the stack contains an IF type cell. This mechanism, thus shields both IF and ELSE, if Unit-1 or Unit-2 happens to be DO-group or BEGIN-block. The following example brings out all the essential points.

```

IF1 E-1 THEN1 L1:DO1;
    S-1;
    IF2 E-2 THEN2 L2: BEGIN1;
        S-2;
        IF3 E-3 THEN3
            IF4 E-4 THEN4 S-3;
/* GENERATION OF ELSE4 AND ELSE3 EXPECTED*/
            S-4;
        END L2;
/* GENERATION OF ELSE2 EXPECTED */
IF5 E-5 THEN5 FORMAT (I1); /* ILLEGAL STATEMENT
    FOLLOWS THEN */
    ELSE5 S-6;
    S-7;
    END L1;
ELSE1 L4:DO2;
    S-8;
    END L4;

    S-9;
/* ILLEGAL ELSE FOLLOWING */
    ELSEx S-9;

```

Note: Subscripts refer to the serial number, given to the IF statement and they are used for showing the pairing of IF-THEN-ELSE. They are not part of PL 7044 Syntax.

E-represent an expression.

S-represents a statement.

Consider the following snapshots,

i)

IF ₄
IF ₃
BEGIN
IF ₂
DO ₁
IF ₁

Before S-4

BEGIN
IF ₂
DO ₁
IF ₁

After S-4

Since top of the stack contains IF₄ and IF₃; ELSE₄ and ELSE₃ would be generated by the FPP.

As can be seen from the stack, DO and BEGIN are shielding the IF statements. IF₂ would be exposed only when a matching END is found i.e., at END L₂; then top of stack would have IF₂ since BEGIN would have been popped off by END statement. Now since next statement is not of ELSE type (IF₅ type), hence ELSE₂ would have been generated, popping IF₂ thereby.

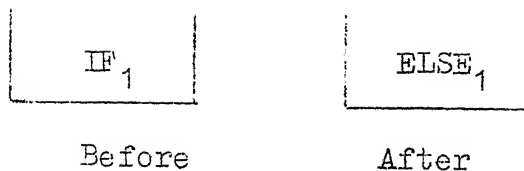
(ii)

THEN ₅
IF ₅
DO ₁
IF ₁

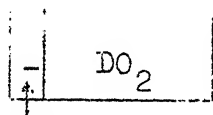
at THEN₅

When FORMAT (I.1); statement consults the top of the stack it will find THEN on the top of the stack. Hence error would be given.

(iii)

at $ELSE_1$

Now the DO_2 will pop off the $ELSE_1$, but it will mark the presence of $ELSE_1$ on the top of the stack so that end label could be generated once the corresponding **END** is found. The stack after processing of DO_2 will be



flag which says that an **END** label is to be generated after corresponding **END** is found.

(iv) After the processing of **END** L4; our symbolic stack would be empty and the occurrence of $ELSE_x$ would constitute an error because there is no matching **IF** in the stack.

This scheme takes care of misplacing of **ELSE** also (to some extent). Consider the case when the cards '**END** L1'; and '**ELSE** L4:DO;' get inter changed, say because of shuffling of the cards, then while trying to process **ELSE** L4:DO; the top of the stack would be found containing DO_1 and it will be sufficient to show the error.

2.9.2 Dummy ELSE and Dummy THEN:

It would be interesting to note that there exist some situations where even shuffling of cards would result in logically correct programs !!

Consider the following two examples,

```
IF1 E-1 THEN1 DO;
      S-1;
      IF2 E-2 THEN2 S-2;
      ELSE2 S-3;
      END;
```

/* ELSE₁ would be generated */

Now interchange the cards END; and ELSE₂ S-3; resulting in the following program.

```
IF1 E-5 THEN1 DO;
      S-1;
      IF2 E-2 THEN2 S-2;
      /* ELSE2 would be generated */
      END;
      ELSE1 S-3;
```

It can be seen though the pairing of ELSE-3 card changes in two examples, both the programs would pass FPP test as far as IF-THEN-ELSE is concerned. A dummy ELSE statement is provided in PL 7044 in the form ELSE; (and also dummy 'THEN', but it is logically redundant) which does nothing more than explicitly defining the pairing of IF-THEN-ELSE.

Had this been done in the example given above, chances that the program remains logically correct even after shuffling would decrease quite a lot.

Let us consider that the programmer wanted his logic to be exactly like that of the first program, but adds dummy ELSE statement, then his program would look like,

```

IF1 E-1 THEN1 DO;
    S-1;
    IF2 E-2 THEN2 S-2;
        ELSE2 S-3;
    END;
ELSE1;    /* Dummy ELSE */

```

Now if the same two cards get interchanged, following program would result.

```

IF1 E-1 THEN1 DO;
    S-1;
    IF2 E-2 THEN2 S-2;
    /* ELSE2 WOULD BE GENERATED */
    END;
ELSE1 S-3;

ELSEx; /* AN EXTRA ELSE, ERROR */

```

Thus FPP would detect the error in this case.

2.10 Classification of END and Multiple Closure:

General format of END statement is as follows:

option -1	END;
option -2	END Label;

Classification of END-statement of option -1 is a very simple job since the cell on the top of stack would dictate the terms, 'END' would be closing a DO-group if the top of the stack happens to be a DO-type cell, otherwise it would have been used to terminate (close) a PROCEDURE/BEGIN block. (Both BEGIN/PROCEDURE blocks get similar treatment from END statement as far as FPP is concerned).

In case top of the stack is not of DO/BEGIN/PROCEDURE type then it can be of IF/THEN/ELSE type. The action taken in the latter situation has already been discussed under the head 'IF-THEN-ELSE Matching'. If stack is empty when the END statement is met, it indicates an extra END and error exit is taken.

If the cell being popped off is flagged to show that an ELSE was shielded by this DO group or BEGIN block, an end label is generated after the code for END-statement.

As for option 2, the name key no. of the label which follows END is taken and is matched against the names stored in the 2nd to 6th words of stack cell. If the name does not match with any of the names stored in the cell or the cell has no

names (no labels used while defining a DO-group or BEGIN block or PROCEDURE block), then that particular cell is given a treatment similar to that in option-1. This closing goes on till either the name matches with some name stored in a cell in which case the process terminates then and there or the stack gets emptied in this process, in which case error condition would be raised since either the name used with END is illegal or END itself is extra.

2.11 DO Predecessor Table and Block Predecessor Table:

FPF while analysing either DO-statements or BEGIN/PROCEDURE statements keeps a count of no. of DO-groups used and the number of blocks used in the program. This information is then conveyed to the second pass and then to the third pass. Together with this, FPF also prepares the predecessor tables viz. DO-predecessor table and block predecessor table. Each of the DO-group and block used in the program is given a unique number, which then serves to identify that particular DO-group or block. The predecessor table stores the identification no. of that group or block (as the case may be) which appears before, lexicographically, the present group or block and which is the closest to it. Consider the following symbolic program :

```

B1: PROCEDURE OPTIONS (MAIN);
  D1:DO;
  D2:DO I=1,2,3 TO 10 BY 1;
    END D1;
  B2:PROCEDURE;
    B3:PROCEDURE (A) RETURNS (CHARACTER);
      D3:DO;
        END B3;
    B4:BEGIN;
      D4:DO;
      D5:DO;
        END;
      D6:DO;
        END B1;
    B5:PROCEDURE;
      END B5;

```

The predecessor tables are as follows:

<u>Block name</u>	<u>B1.No.</u>	<u>B.P.N.</u>	<u>DO name</u>	<u>DO no.</u>	<u>D.P.N.</u>
imag	0	0	D1	1	0
B1	1	0	D2	2	1
B2	2	1	D3	3	0
B3	3	2	D4	4	0
B4	4	2	D5	5	4
B5	5	0	D6	6	4

A DPN (DO predecessor no.) of 0 indicates that there is no DO group enclosing this DO group. It is assumed for logical completeness that there exist an imaginary block which encompasses all the external procedures and built in procedures. It is given a block no. '0'.

These tables in fact preserve the information about lexicographical relationship among various blocks and DO groups. This information is used very often for arriving at various logical decisions regarding scope of names, validity of a reference, sharing of first order storage area etc. These would be explained in the following chapters.

2.12 Separation of Specifications and Declarations from the Text:

The PL 7044 compiler puts one more restriction on PL/I language. All the declare statements must be placed immediately after the BEGIN, PROCEDURE or ENTRY statements, as the case may be. Error condition is raised if a DECLARE statement is found at any other place. Further after an ENTRY statement, only specifications can be put. No declaration should occur after an ENTRY statement.

FPP detects the changeover from the declaration and specification phase to normal text phase and it generates a message for second pass to this effect.

FIRST PASS PROCESSOR Flow Charts

NOMENCLATURE:

1. Attribute Sets:

SET-1:	BEGIN, END, DO, STOP, ELSE, EXIT
SET-2:	FLOW, NOFLOW
SET-3:	SET-3A U SET-4
SET-3A:	GO, GOTO, CALL, CLOSE, GET, OPEN, PUT, READ, WRITE, DO, DECLARE, IF, ELSE
SET-4:	ALLOCATE, DELETE, FETCH, FREE, LOCATE, ON, RELEASE, SIGNAL, UNLOCK, REVERT, REWRITE
SET-5:	DEFAULT, DELAY, WAIT

2. Lexical Units:

IDENT:	Identifier or non keyword
INT:	Integer
Name:	Identifier or Keyword
Operator:	See Appendix-A1
PRW:	Pseudo Reserved Word or Keyword

3. Routines:

ADCEL.:	Add one six word cell on the top of the program stack.
ADCEI:	Classify the top most cell according to the call (IF/THEN/ELSE/DO/BEGIN/PROCEDURE).
BLKCLS:	Close the block, presently open and update various tables and flags.
ENDSPC:	Check that the specifications and declarations are over for the block

ENDSPT: Initialise various flags so that DECLARE statement can occur, for the new header statement (PROCEDURE/BEGIN/ENTRY).
 ENTBUF: Enter the lexical unit in the buffer.
 GENELS: Generate a dummy ELSE statement.
 GETUNT: Get the next lexical unit.
 OUTCOD: Output the type of statement.
 OUTPUT: Output the body of the statement collected in the buffer.
 PUTLBL: Enter the statement-labels prefixed to the statement in the label-chain and generate a constructed label constant in lieu of them.
 REMCL.: Remove the top most cell from the program stack.
 REMCEL: Generate a success-label, if the top most cell is of THEN type.
 Generate a false-label and end-label (at the end of current statement), if the top most cell is of ELSE type.
 REMCL1: Generate a success-label, if the top most cell is of THEN type.
 Shield the ELSE, if top most cell is of ELSE type.
 RESET1 : Initialisations for a new statement.
 RESET2 : Initialisations to be done if a label definition is obtained.
 TSTCEL: Raise an error condition, if top of the stack is of THEN or ELSE type.

4. Symbols and Flags:

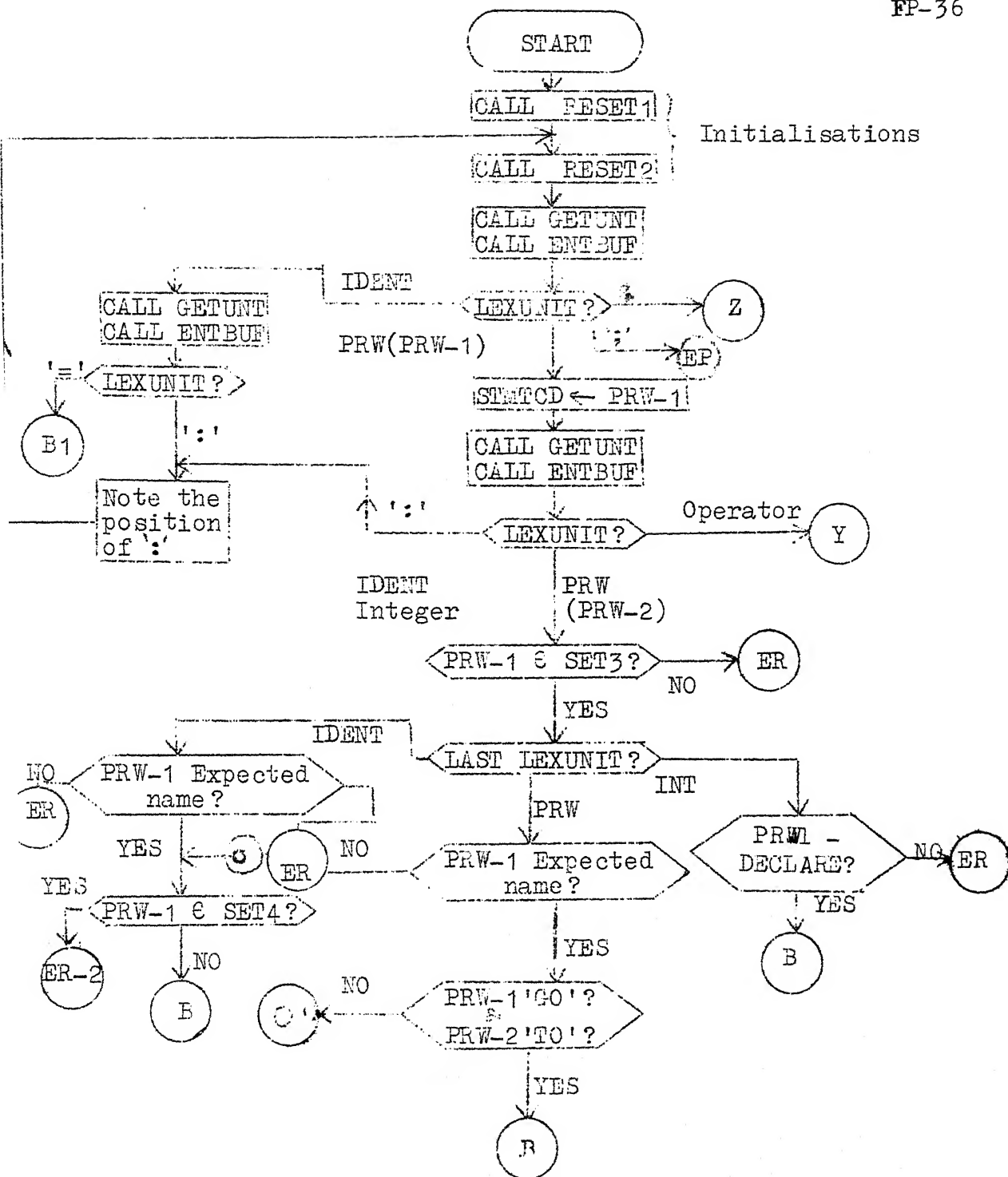
ASGNCD:	Code for assignment statement.
CNTSTM:	Represents the continuation of the same statement (i.e. after THEN/ELSE)
DOFIG:	Flag used to distinguish between an END closing a DO-group and an END closing a block.
FNDEQ:	Flag to decide the presence of '=' outside parentheses.
LBLINT:	'Label Initialisation' flag.
NXTSTM:	Represents the start of a new statement.
ONEBLK:	Flag to distinguish between an END closing one group/block and an END effecting multiple closure.
PRNCNT:	Differential count of parentheses

5. Error Exits:

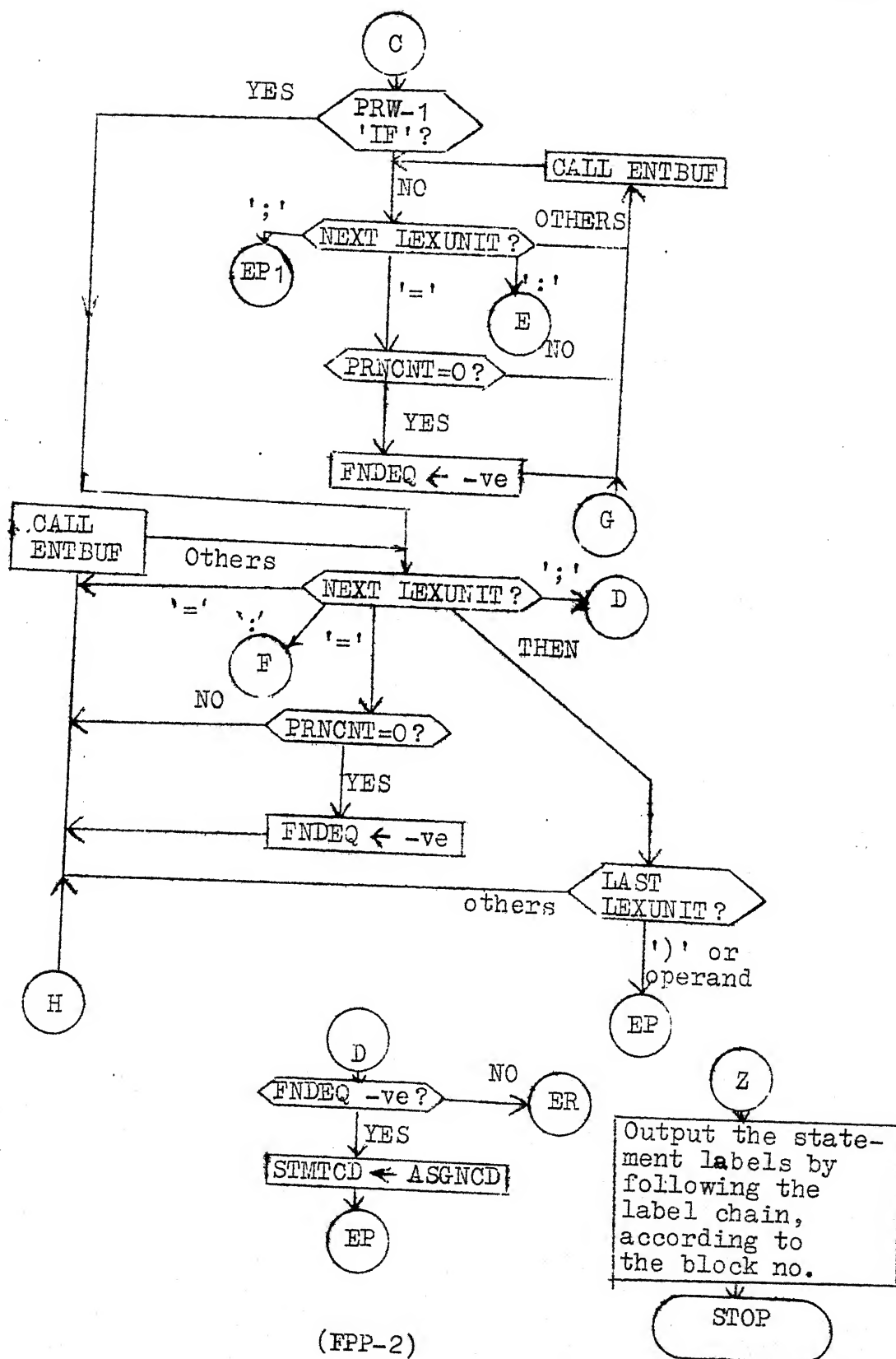
(Action taken - skip to the next statement)

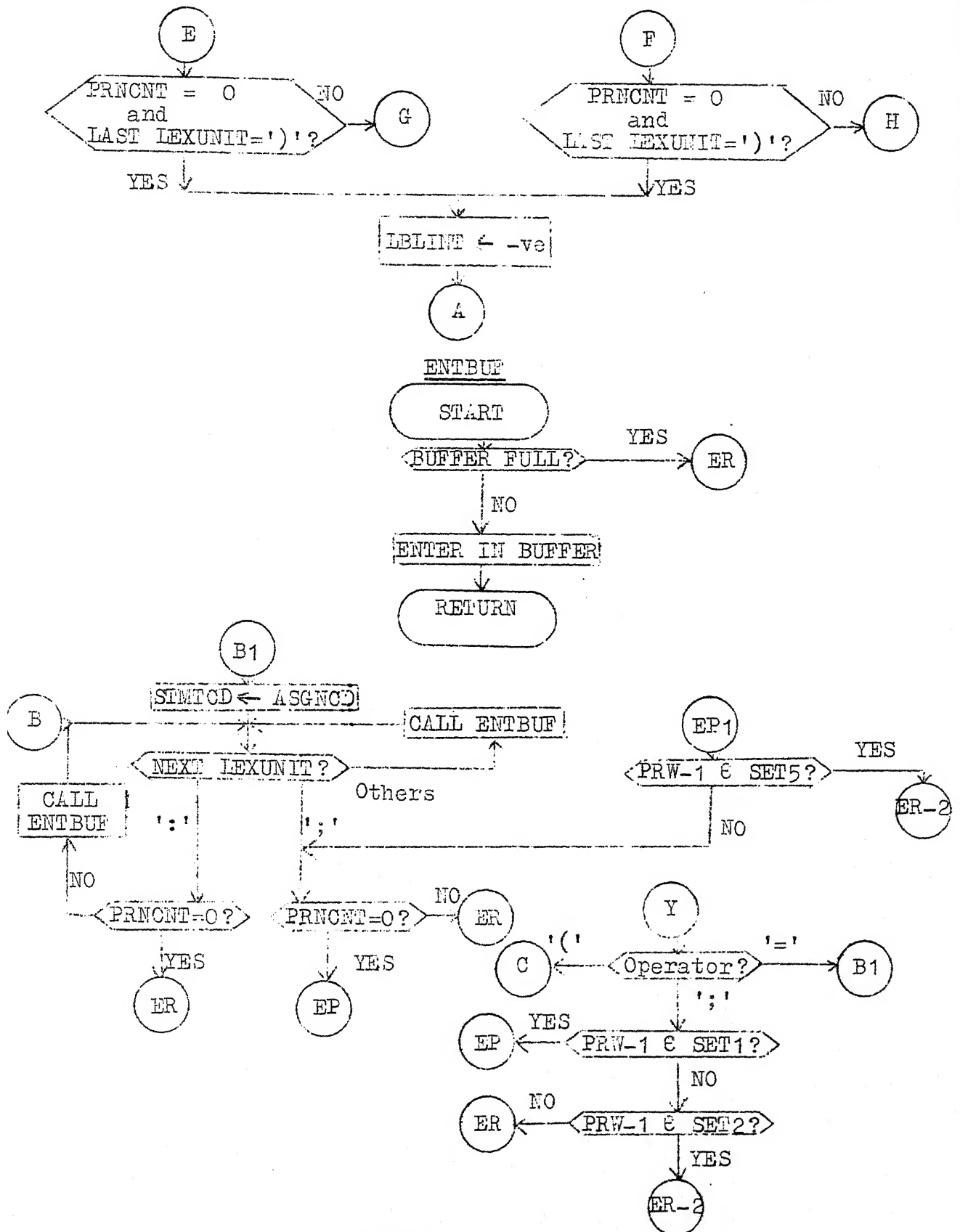
ER:	Miscellaneous errors
ER-2:	Illegal statements (Not allowed in PL-7044).

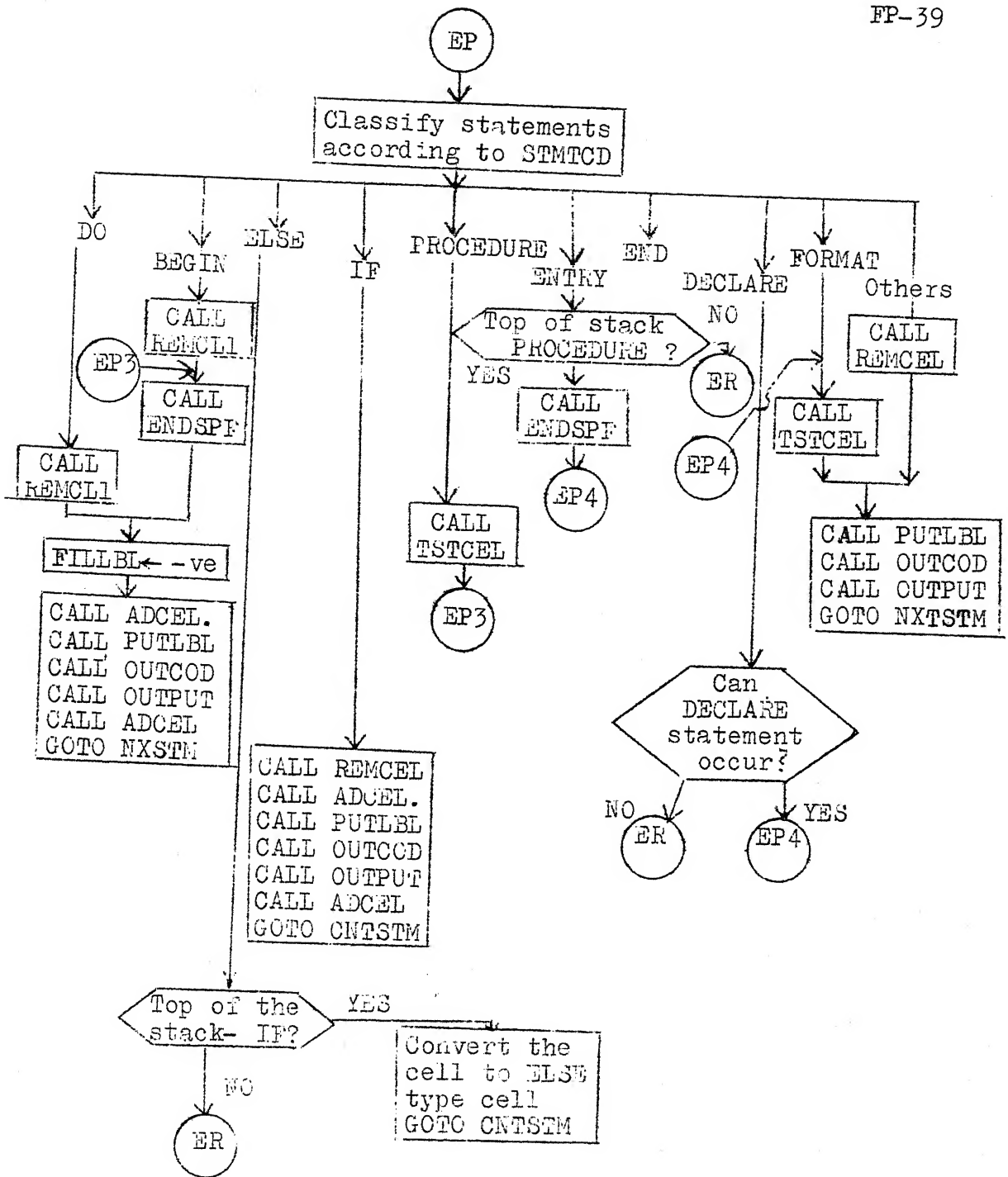
- Note:
1. Lexical units for which a branch is absent in a decision box, constitute error.
 2. Extended lexical activities have not been shown in the flow chart.



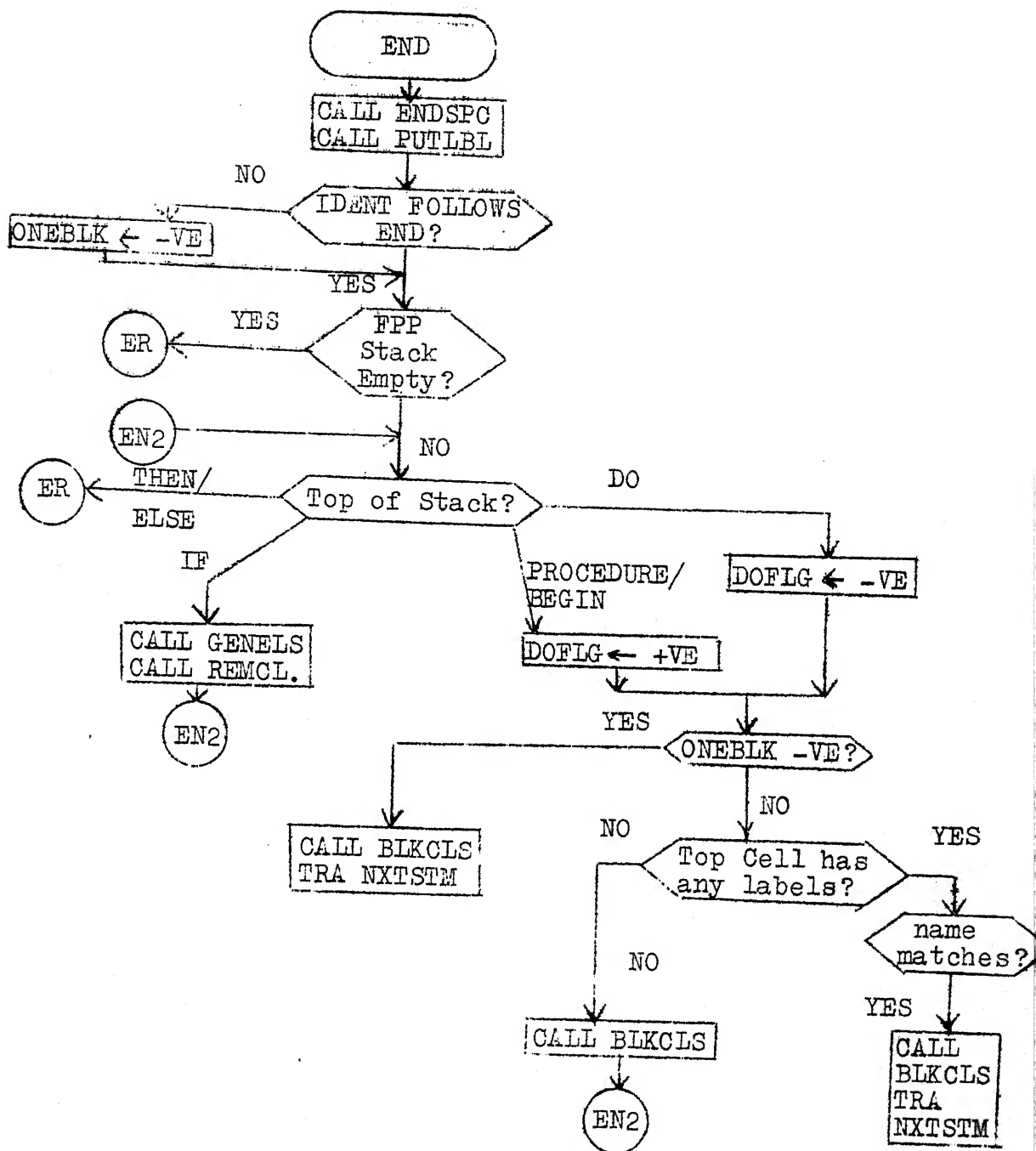
(FPP-1)







(FPP-4)



(FPP-5)

1. Declarations and Declare Statement:

1.1 Declaration:

An identifier in a PL/I program may refer to one of many classes of objects. For example, it may represent a variable referring to a complex number; it may refer to a file; it may represent a variable referring to a character string; it may represent a statement label or represent a variable referring to a statement label.

In a program a given identifier may be established in different parts of the program as different names. For example an identifier may represent an arithmetic variable in one part of a program and an entry constant in another part. However, these parts can not overlap. The recognition of an identifier as a particular name is established through declaration of the name. Declarations may be explicit, contextual or implicit.

1.1.1 Explicit Declarations:

Explicit declarations are made through use of the `DECLARE` statement, label prefixes, and specifications in a parameter list.

1.1.2 Contextual Declarations:

The syntax of PL/I allows unqualified identifiers appearing in certain contexts to be recognised as a name without an explicit declaration. Contextual declarations can occur with Builtin procedure names and file names.

1.1.3 Implicit Declaration:

An identifier that is neither explicitly declared nor contextually declared is declared implicitly. Other attributes assigned to an implicit declaration depend on the initial character of the identifier.

For details about scope of a declaration, explicit declaration, contextual declaration, implicit declaration, establishment of declarations, syntax rules of DECLARE statement etc., the reader is referred to the PL/I language specifications.

1.2 Declare Statement:

Declare statement is a non executable statement that explicitly specifies attributes which are to be associated with a name. A declare statement may have a label prefix. On transfer of control to such a label, the label is treated as if it were on a null statement and execution continues with the next executable statement. The general format, syntax etc. is shown in Figure DC-1.

The routine which analyses the declare statement would be explained with the help of the following example:

```
DECLARE A REAL, B, 1 C STATIC, 2 D,
      3(E REAL, I FIXED), 2 E, 3 F,
      (8 G, 7 (H REAL, I)(10),J LABEL), 4 K REAL),
      2 H, ((X,Y) REAL, K) DECIMAL, L LABEL)
      STATIC EXTERNAL;
```

General Format:Non factored:

```
DECLARE [level] identifier [attribute] ...
      [, [level] identifier [attribute] ...] ...;
```

Factored:

```
DECLARE declaration list;
```

where "declaraction list" is defined as declaration
 declaration [, declaration] ...

where "declaration" is defined as:

```
[integer] { identifier | (declaration list) }
[(dimension-attribute)] [attribute...]
```

Examples:Non Factored:

```
DECLARE ARRAY (10), 1 A, 2 B REAL, 3 C, 2 E;
```

Factored:

1. DECLARE ((A FIXED, B FLOAT) STATIC, C) EXTERNAL;

The above declaration is equivalent to the following
 declaration

```
DECLARE A FIXED STATIC EXTERNAL, B FLOAT STATIC EXTERNAL,
      C EXTERNAL;
```

2. DECLARE 1 A AUTOMATIC, 2 (B FIXED, C FLOAT, D CHAR(10));

The declaration is equivalent to the following
 declaration:

```
DECLARE 1 A AUTOMATIC,
      2 B FIXED,
      2 C FLOAT,
      2 D CHARACTER (10);
```

1.3 Organization:

The declare statement has three modules which are called depending upon the type of declaration being processed. Type-of-declaration refers to one of the following:

- i) Ordinary or non structure declaration,
- ii) Structure declaration, and
- iii) Factored declaration -
 - a) Factored ordinary
 - b) Factored structure

In the above example, three types can be identified as follows:

Ordinary declaration-

A REAL, B,

Structure declaration -

1 C STATIC, 2 D, ..., 2 H,

Factored declaration -

a) Factored Ordinary -

((X,Y) REAL ...) STATIC EXTERNAL;

b) Factored structure -

3(E REAL, I FIXED), and

(8 G, 7((H REAL ... REAL),

The declare routine has one coordinator to coordinate the activities of these modulus. Depending upon the type of the lexunit under inspection, the coordinator calls the appropriate

module. Once a module is through with its work it returns control to the coordinator which asks for the next lexunit and the whole cycle is repeated till a ';' is met. At ';' control is returned to the main executive.

The main job entrusted to the declare statement is the construction of the symbol table. To do the attribute analysis, declare routine calls various Attribute Analysis Routines (AAR) which assign storage, generate output, if any, for doing array allocation and initialisation & generate prologue list entries in case of formal parameters.

AARs have been discussed in section 3 of this chapter.

1.4 Description of the Modules:

A short description of the three modules follows.

1.4.1 Ordinary Declaration:

Appearance of a 'name' signals the presence of ordinary declaration and ordinary-declaration module is called. This module enters the symbol in the symbol table by calling the ENTER routine and then calls the AAR(ATORDN), which analyses the attributes for ordinary declaration and finally returns control to the coordinator.

1.4.2 Structure Declaration:

Presence of an integer as the first lexical unit in the new sub-field indicates the declaration of the structure.

Coordinator in this case calls the structure declaration module. This module checks that the integer should be '1', since a major structure can have a level number of '1' only. As usual this symbol is entered into the symbol table as major structure. Structure number maintained by the compiler is increased by one. This structure is then referred to by this structure number throughout the compilation phase. AAR (MJSTRC) is called which does various initialisations for structure declaration in the attribute analysis routine. It also finds out about the scope and storage attributes those would be applicable to the entire structure, since scope and storage attributes can be declared with major structure name only. In case of a formal major structure, a prologue list word is also outputted.

For processing a structure declaration, a stack is used which has one entry for each of the items declared in a structure. Each factorisation is considered to be one item for this purpose. Since the level numbers used by the programmer need not be continuous, the compiler assigns internal level numbers, which are continuous. Consider the following example:

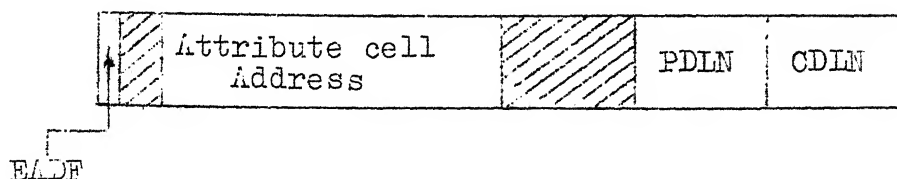
```
DECLARE 1 A, 3 B, 10 C, 2 D, 15 E, 20 F, 10 G;
```

The level numbers assigned by the compiler would correspond to the following declaration:

```
DECLARE 1 A, 2 B, 3 C, 2 D, 3 E, 4 F, 3 G;
```

Internal level numbers become necessary to bring uniformity among various structure declarations so that one number corresponds to one unique level.

The format of the cell of the structure-stack is as follows:



where,

PDLN : Programmer Defined Level Number

CDLN : Compiler Defined Level Number

EADG : Explicit Attributes Defined Flag

= -ve if attributes are defined explicitly

= +ve otherwise.

For entering a minor structure name or a element name in the symbol table, covering item address and compiler defined level number are needed. To determine the covering item address, stack is searched from top to bottom. At each cell, the PDLN stored is compared with the PDLN of the present item. The first item which has a PDLN less than the PDLN of the present item is the covering item. The address stored in that cell is then taken as the covering item attribute cell address. The CDLN of the present item is then one more than the CDLN stored in the cell found.

EADF is needed for carrying out the following checks:

1. Array of structures are not used, since they are not provided in PL-7044.
2. Level numbering is all right i.e. the covering item has no explicit attribute definition .
3. To assign default attributes to the predecessor item if necessary.

EADF is put 'on' by the AAR if it finds explicitly defined attributes for an item in the structure definition. If an item has no explicitly defined attributes (including the dimension attribute), the AAR gives it a major type of 'minor/major structure' and returns. The routine called for assigning attributes to minor structures/base elements is MNSTRC.

If the CDLN of the preceding cell (lexicographically) is more than or equal to the level number of the present item and EADF is off, default attributes are assigned to the preceding item and EADF is put on.

On the other hand if CDLN of the preceding item is less than the CDLN of the present item and preceding item has EADF on, error condition is raised. This error can be either because of dimension attributes given to the minor structure or because of illegal level numbers used.

Factorisation in Structure Declaration:

Only base elements can be factored inside the declaration of the structure. Following declaration is illegal -

```
DECLARE 1 A, 2(B,C), 3 D;
```

This restriction comes because a factorisation is treated as one unit. All the symbols occurring inside the factorisation are assigned attributes, whether explicit attributes are declared or not. The factorisation in a structure declaration can be of two types. First type of factorisation can be said to be 'level number factorisation'; 3(E REAL, I FIXED) represents this form of factorisation. In the second form of factorisation level numbers are not factored, in general. In fact second type may consist of first type of factorisation also as shown in Fig. DC-1. There is a restriction on the level numbers used inside the factorisation. All the numbers used inside the factorisation should be greater than the PDLN of the item immediately preceding the factorisation. Further inside the factorisation, the PDLN should either be same or be in monotonically decreasing order. The above two restrictions ensure that all these items form sister nodes or graphically they are at the same level. For both kinds of factorisation, the covering item is same for all the items since graphically they are at same level.

The definition of a structure is terminated when after ',', either a symbol appears (showing ordinary declaration) or integer '1' appears (showing another structure) or a symbol appears immediately after one or more left parentheses (showing factored declaration of ordinary type). The main example in Fig. DC-1 corresponds to the last case.

At the end of a structure definition, the last item is given default attributes, if its EDAF is not 'on'. (So is the case in the main example. The item 'H' is assigned default attributes in this case).

Before returning control to the coordinator, the structure stack is emptied.

1.4.3 Factored Declaration:

Factored structure declaration has already been discussed in the structure declaration module. In fact structure declaration module calls this module when it needs a factorisation.

This module keeps a differential count of left and right parentheses (PRNCNT). PRNCNT is increased by one for every left parenthesis and is decreased by one for every right parenthesis. When PRNCNT reduces to zero, factorisation is considered to end.

The AAR maintains a stack, called 'factor stack', for analysing factored declarations. At every left parenthesis a

LP-type cell is added on top of the factor stack. It is needed to match the right parenthesis when one is obtained so that range of factorisation could be determined. A detailed description of factor stack is available in the Section 3.

The factor-declaration-module calls different AARs in different contexts, viz.,

- FCORDN : for ordinary factorisation after the symbol
e.g. ... (X REAL ...
- FCSTRC : for structure factorisation after the symbol
e.g. ... 2(X REAL ...
- RPORDN : for ordinary factorisation after the right
parenthesis e.g. ..., (Y,X) REAL ...
- RPSTRC : for structure factorisation after the right
parenthesis e.g. ..., (X,Y) REAL ...

This distinction of routines to be called simplifies the job of the attribute analysis routine.

Control is returned from this module after PRNCNT has been reduced to zero and post right parenthesis attribute processing routine (RPORDN/RPSTRC) has been called.

2. Symbol Table:

2.1 Introduction:

Symbol table is a repository of information about a symbol and as such is consulted by all the routines for extracting various kinds of information about the symbol. The symbol table takes up exactly at the point where name-table of First Pass Processor (FPP) had stopped. The name table supplies a unique 'name-key-no.' to each of the symbols used in the program. Throughout the syntactic analysis and semantic interpretation phase, the symbol is referred to by this 'name-key-number'.

The organisation of symbol table proposed and implemented. automatically takes care of the block structure of the PL/I language. As and when a new declaration is made in a new block, an additional cell is added to the symbol table and till the time either a new declaration of same name is made in another block, or the block in which the declaration has been made is terminated, this declaration would be accessible to the enquiring routines. The organisation of the symbol table takes care of the default declarations also without any difficulty. If a symbol does not exist in the symbol table and call is not from I/O routines, it is given default attributes of arithmetic type, otherwise it is treated as contextual declaration of files.

Except in the case of label variables, the validity of reference is taken care of in the compilation stage itself so that run time overheads are minimised. Consider the following program:

```

MAIN: PROCEDURE OPTIONS (MAINS);
      DECLARE LABEL LABEL;
      .
      .
      .
      BLOCK1:BEGIN;
      HELL: LABEL=HELL;
      .
      .
      .
      END BLOCK1;
      GO TO LABEL;
      ENL MAIN;

```

In this example the GOTO statement will raise the error condition, if executed, because it is trying to enter into a closed block. However, it can not be detected at compile time.

All the builtin procedures and pseudo variables are preloaded in the symbol table and they belong to the imaginary block, encompassing all the external blocks. This ensures that they are never removed from the symbol table since the imaginary block is considered to be open throughout the program.

2.2 Organisation:

The FPP determines the total number (N) of symbols used in the program. (N includes all the pseudo reserved words, builtin procedures and pseudo variable names, though all of them

may not have been used as a symbol or name of a variable. This number is supplied by the 'fixed-table' which keeps all these names.). N is transmitted to the second pass executive, which takes an N word array from the free core area and makes the 'index-table'. Index table has one word for each of the names used in the program, and for each of the keywords, pseudo variable names and builtin procedure names. Attribute cell address of all the builtin procedures, and pseudo variables is filled in the corresponding word of the 'index-table' so that they are available all the time. Name-key-number assigned to each of the names used by the FPP is then analogous to the offset in the index-table. A predicate P can be defined as follows:

$$P(\text{Symbol with 'name-key-number' } x \text{ is in symbol table})$$

$$= 1$$

$$\iff \text{INDEX}(x) \neq 0$$

This predicate is made use of to determine whether a particular name exists in the symbol table.

Index-table points to the latest attribute cell for a name. (Previous attribute cells for the same name corresponding to declarations in the predecessor blocks are linked backwards in themselves). The attribute cell consists of six words, taken from the 'free-list' (Appendix-P). The format of symbol table entry is given in Appendix - D.

2.3 Service Routines of Symbol Table:

From time to time, the symbol table has to attend to the requests from various routines for various kinds of favours. For example a call to the symbol table can be either for adding a cell or for deleting a cell. Sometimes a call might be made for changing some flags etc. The routines doing all these jobs are as follows:

ENTER

LOOKUP

ENTLBL

WIPSTB

ADDWDP

INBPWA

INBPWT

INBLUN

A short description of each of these routines follows.

2.3.1 ENTER:

This routine is called for entering a symbol in the symbol table. It is called by the declare routine and the label entering routine (ENTLBL). The ENTER routine is made general enough to take care of the declarations of structures also. For entering a symbol in the symbol table, a six word cell is taken from the free-cell-list and is added at the bottom of the attribute list. It also sets various pointers but filling of

attributes is done by the calling routine. (In case of declare-statement it is done by the attribute analysis routine whereas the the label-entering routine does this work on its own).

2.3.2 LOOKUP:

This is the most widely used routine out of the whole set. It returns the cell address of the latest attribute cell for the symbol. This routine takes care of qualified names also. The lookup routine is so designed that it automatically assigns default attributes to a symbol which is found to be non existing in the attribute list. This declaration belongs to the EXTERNAL procedure, that is open at that time. If the calling routine had expected some thing other than an arithmetic data variable/file name, it is the duty of the calling routine to give out 'error'.

Both these routines viz. ENTER and LOOKUP raise the error condition in case of illegal qualifiers, ambiguous qualifications, non existing structure references etc.

2.3.3 ENTLBL:

This routine enters statement-label-definitions in the symbol table. Statement-label-definitions are collected by the first pass processor and are sorted according to the block serial number. At the time of opening a block all those statement-label-definitions, which belong to the block being opened, if any, are loaded in the symbol table.

In case of procedure and entry names (primary and secondary entry constants respectively) the symbol table is to be supplied the information about the type of result being returned also so that proper code could be generated, along with conversions if any, in case of function-reference. This information about the type of result returned can be explicitly defined by specifying RETURNS specification in the PROCEDURE or ENTRY-statement. In such a case the code for RETURNS is transmitted along with the statement-label-definitions by the FPP. It is analysed by the attribute analysis routine and this information is entered into the symbol table. In the absence of RETURNS specification, default attributes are assigned depending upon the first character of the name.

This routine generates the 'procedure header word' (Appendix B) also. The address of this word is filled in the symbol table.

2.3.4 WIPSTB:

To take care of the scope of name in the symbol table itself, it becomes necessary to delete all the declarations made in a block at the time of terminating that block. This routine does exactly this work only and it returns all the cells belonging to the block being terminated to the free list. All the declarations, that were known before this block was opened, become active once again due to this activity. The index table is updated to point, now to these cells once again.

All the default declarations which were made in this block are hooked to the predecessor block since they belong to the EXTERNAL procedure and as such they remain active as long as the EXTERNAL procedure remains active, unless declared once again in a contained block.

2.3.5 INBPWA, INBPWT, INBLUN, ADDWDP:

These four routines are used for builtin procedure initialisations. INBPWA is for initialising builtin procedures with arguments. INBLUN finds out the most commonly use builtin procedure with a particular name. ADDWDP is for adding a cell with default attributes. INBPWT initialises the builtin procedures without arguments.

2.4 SYMBOL TABLE FORMAT:

Symbol table format is shown in Appendix - D. However, description of various pointers, their meaning and their need follows.

2.4.1 Pointers, their Meaning and their Need:

(1) Last in stack (LSTCK):

It is the address of the attribute cell which immediately precedes it in the attribute list. This is needed for traversing through the attribute list backwards. Forward traversing is not needed in general, except in the case of structures (where forward pointer is also provided),

(2) Index table offset (INDXOF):

It is the name-key-no. of the symbol. It is needed for updating the index table at the time of terminating a block. Also this unique name-key-no. serves as the 'name' of the symbol and is used in lieu of the actual name (which has no existence in the second pass). Routine analysing the BY NAME option makes extensive use of this pointer since its operation depends on matching names only. Routine analysing the 'LIKE' attribute also makes use of this pointer.

3. Last in Stack with Same Name (LSTSNM):

All the attribute cells for variables with same name are linked backwards. LSTSNM points to the last cell with same name. This pointer is needed for updating the index table at the time of terminating a block as well as for determining the proper reference in case of both qualified and non-qualified names.

Consider the following examples:

```

DECLARE 1 E, 2 C, 3 F, 3 H, 4 F, 2 G, 3 H;
DECLARE 1 A, 2 A, 3 A, 4 A, 2 B, 3 A, 4 C, 5 E, 2 D,
        3 D, 3 E, 4 A;
DECLARE B;

```

If a lookup call is given for the qualified name 'B.A' symbol table will return the cell which is identified by 'A.B.A'. Lookup routine will first consider B and would initiate a search for B. The first cell to be hit for B, corresponds to the scalar definition of B (since it was declared last). Since a '.' follows

B, this definition would be ignored. Now a backward search starts along LSTSNM, and in next trial it hits another B which is at a level greater than one and it may be taken if further tests succeed. Since there is no other cell with name B, (LSTSNM=0) this cell would be taken.

Search Strategy for the First Name of the Qualified Name:

The strategy followed in the backward search for first name of the qualified name is that either it terminates at a major structure (level=1) or at a minor structure, if it can be identified unambiguously. For example a call for lookup of 'C.E' will turnout to be ambiguous since 'C' can not be identified uniquely as not enough qualification is provided for distinguishing between '(E.)C' and '(A.B.A)C' whereas a search for 'D.E' will result in success since 'D' can be identified uniquely.

Search Strategy for an Unqualified Name:

A search for an unqualified name stops either at a major structure (level = 1) or at a non structure variable if one is found. In case no such definition exists in a block in which some other definition exists at level greater than one, the search stops at an element which is at the highest level (lowest level number) in a structure provided there is no other entry (with same name) at level greater than one in the same block but in some other structure. For example a Lookup call for 'B' will

terminate the search at the scalar definition of 'B', whereas a call for 'E' will terminate at the major structure 'E', though a minor structure definition exists for 'E', which would be met before the former. However, a Lookup call for 'C' will turn out to be ambiguous because of the same reasons as for the failure of 'C.E'.

Search Strategy for the Rest of the Qualification:

There are two methods for completing the LOOKUP of a qualified name once first name has been identified uniquely -

- i) Top to bottom search, and
- ii) Bottom to top search.

In top to bottom search we start from the cell which was looked-up last time and follow the forward chain there onwards and go upto the range of the minor/major structure definition which was looked up last time. During this travel that element is selected which is at the highest level in this minor/major structure, if one could be found unambiguously.

In bottom to top search, the backward pointer is followed from the cell corresponding to the latest definition of the name presently under consideration till the cell looked up last time is reached.

This compiler employs the first method.

Search is initiated at the next to the one looked up last time and is terminated when the range of the minor/major structure

ends. At each step the name is matched with the name stored in the cell to determine whether they are same. If the names don't match, search continues, otherwise if no other definition of the same name under this minor/major structure has been found till now, it is noted and search is continued. However if one definition with same name has already been found when this cell is found, the level of the already found item is compared with that of the present cell. If old cell is at a higher level, it is retained and search continues. If old cell is at the same level, a flag is put on to indicate the presence of two elements with same name at same level (but covering elements would be different) and search is continued. However if old cell is at a lower level, the present cell is taken and the flag is reset. To speed up this process an extra test is performed at each step (when the names match, of course). If the level number of the present cell is one more than the levelnumber of the item looked up last time, the present cell is taken and search is terminated. This is possible since ENTER routine makes sure that two elements of same name at same level do not have same covering item.

At the end of the search if flag is found 'on', it indicates ambiguous qualification. If no cell is found at the end of the search, it indicates illegal qualification. Consider the LOOKUP call for 'A.A'. First 'A' would be found to be the major structure according to the strategy already discussed. The search for second 'A' would stop at the next 'A' itself since

it is at level $2(1+1)$ i.e. one more than the level of first A. However in the case of 'A.E', search would stop only at the end of structure 'A' and '(A.D.)E' would be the result of the search. ('(A.B.A.C.)E' would have been ignored since it is at a lower level than '(A.D.)E'). A look up call for 'E.H' would be ambiguous because LOOKUP routine can't decide between 'E.C.H' and 'E.G.H' as not enough qualification is provided.

4. Covering Item Address (COVER):

It is the address of the cell corresponding to a major/minor structure which is one level higher and which contains this item (parent node). It is needed for detecting ambiguous declaration of a structure. Such a structure is shown in the next section.

5. Last in Structure with Same Name (LSRSNM):

It is the address of the cell with same name and declared in the same structure. This pointer is logically redundant because it can be dispensed with but at the cost of more complexity and increased search time. Its effect can be simulated with LSTSNM and structure number. It has been added because space was available in the symbol table.

LSRSNM is used to detect ambiguous declaration of structure. In particular, ENTER routine checks that two base-elements or minor structures (at same level) with same name

do not have same covering item also. In case the covering item happens to be the same, error condition is raised. For an example consider the case.

```
DECLARE 1 A, 2 B, 3 C, 4 D, 3 D, 4 D, 2 B;
```

This declaration of a structure is not valid since there are two minor structures ('B', at Level 2) having the same covering item 'A'. This error would be detected while the second B would be getting declared because in that case LSSRNM would point to first B and both of them would be having the same covering item address in COVER. This search is followed till LSSRNM becomes 0, which indicates that there is nothing more in this structure with same name as the one getting defined.

This should not cause a misunderstanding that in a structure two (or more) minor structures or base elements at same level can not have same name. Only thing that is important is that they should not have same covering item. In the example given there are two items with same name (D) at same level (4) but their covering items are 'A.B.C' and 'A.B.D' respectively.

6. Next on Structure (NXTSTR):

It is the address of the next item (lexicographically) in the structure. If there is no next item, it is made zero. It is used to traverse the structure in the forward direction. It is used by the routine analysing the 'LIKE' attribute and many other routines including the LOOKUP routine.

7. Next in Structure at Same Level (NXSTLV):

It is the address of the next cell at the same level and having same covering item. It is used by the 'BYNAME' routine, which analyses the BYNAME option.

The representation of attribute cells for different data items is given in Appendix D.

3. Attribute Analysis Routines:

3.1 Introduction:

An identifier in a PL/I program may refer to one of many classes of objects. Those properties that characterise the object represented by the name, and the scope of the name itself, together make up the set of attributes that are to be associated with the name. There are a number of classes of attributes. When an identifier is used in a given context in a program, attributes from certain of these attribute classes must be known in order to assign a unique meaning to the identifier. For example if an identifier is used as a data variable, the data type must be known; if the data type is arithmetic, the mode and scale (fixed point or floating point) must be known.

Attributes are given to identifiers by explicit, contextual and implicit declarations. In addition attributes may be given to identifiers by the standard default rules (Appendix G). Attributes are specified in a DECLARE - statement and they follow the identifiers to which they refer (non factored

declaration). Attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute for many identifiers.

3.2 Classes of Attributes:

Attributes have been classified in four major categories:

- a. Storage class and scope attributes,
- b. Data attributes,
- c. Miscellaneous attributes, and
- d. 'No-action' attributes.

3.3 Attribute Analysis Routines:

The ATTRBT deck consists of following eight routines for attribute analysis.

For Non-Structure or Ordinary Declarations:

1. ATORDN: For attribute analysis of non-factored ordinary declarations e.g.
DECLARE A REAL, B FIXED (5,0) STATIC;
2. FCORDN: For attribute analysis of factored, ordinary declarations, after a name e.g.,
DECLARE (A FIXED, B)(10) STATIC;
After the names A and B, FCORDN would be called to do the attribute analysis.
3. RPORDN: For attribute analysis of factored ordinary declarations, after the right parenthesis e.g.
After ')' i.e. (10) STATIC would be analysed by RPORDN routine.

For Structure Declarations:

4. MJSTRC: For attribute analysis of major structure name
e.g.

```

DECLARE 1 A STATIC, 2(B FIXED, C FLOAT) DECIMAL,
        2 D, 3 E CHARACTER;

```

MJSTRC would be called after the major structure name A.
5. MNSTRC: For attribute analysis of minor structure/base elements, (non factored) e.g.
After E in the above example MNSTRC would be called to do the attribute analysis.
6. FCSTRC: For attribute analysis of factored base elements, after the name e.g.
FCSTRC would be called after the names B and C.
7. RPSTRC: For attribute analysis of factored base elements after the Right Parenthesis (RP) e.g.
After ')' i.e. at DECIMAL, RPORDC would be called.

For formal parameter common to PROCEDURE and ENTRY statement:

8. PUTFML: When a parameter is shared in PROCEDURE and ENTRY statement, an entry in the prologue list of procedure statement would have been generated and storage would have been assigned. The prologue list corresponding to the secondary entry point, should also have an entry for this parameter and should refer to the same storage. PUTFML is called for this purpose.

For Default Attribute Assignment:

For assigning default attributes to normal variables, ordinary/base elements, the calling sequence is as follows -

MSM PUTDF

TSX ATORDN/MNSTRC,4

Same calling sequence is to be followed to generate prologue list entry for a parameter which doesn't appear in a DECLARE statement.

All the above routines share a main module called '.ATTRB' which would be described now.

3.4 Main Module:

The operation of the main module of the Attribute Analysis Routines (AAR) can be divided into four phases.

- i) Collection of the attributes,
- ii) Making the 'complete attribute set',
- iii) Assigning storage and generating output, if any, and
- iv) Filling the symbol table entries.

The main module would be described in terms of these four phases.

3.4.1 Collection of the Attributes:

Main module has an array of five words called Representative Words (RW) for collecting the attributes. The first RW stores the attribute cell address of the symbol for which

attribute analysis is being done. Rest of the four words correspond to the four major classes of the attributes as described in earlier section. These words are initialised in the beginning and at the end of the collection phase, if any of the word is found to be in the initialised state, it implies that no attribute of that class has been specified. However, the sign position of the particular RW being negative indicates the presence of at least one attribute of that class.

Each of the attribute has been given a unique code number which identifies its major class, minor class, first subclass, second subclass etc. The attributes of one major class are divided into various minor classes, which are further subdivided into various subclass and so on. The hierarchical structure of some of the major classes is shown in Figure DC-3.

This classification simplifies the identification of an attribute. Consider the code words of some of the attributes.

	M_J^C			$II-S_C$	$I-S_C$	M_N^C	
	0	1	2	3	4	5	(Character pn.)
REAL	2	0	0	0	1	1	
COMPLEX	2	33	1	1	1	1	
DECIMAL	2	0	0	2	2	1	
STATIC	1	1	0	0	1	1	
EXTERNAL	1	1	0	0	2	2	

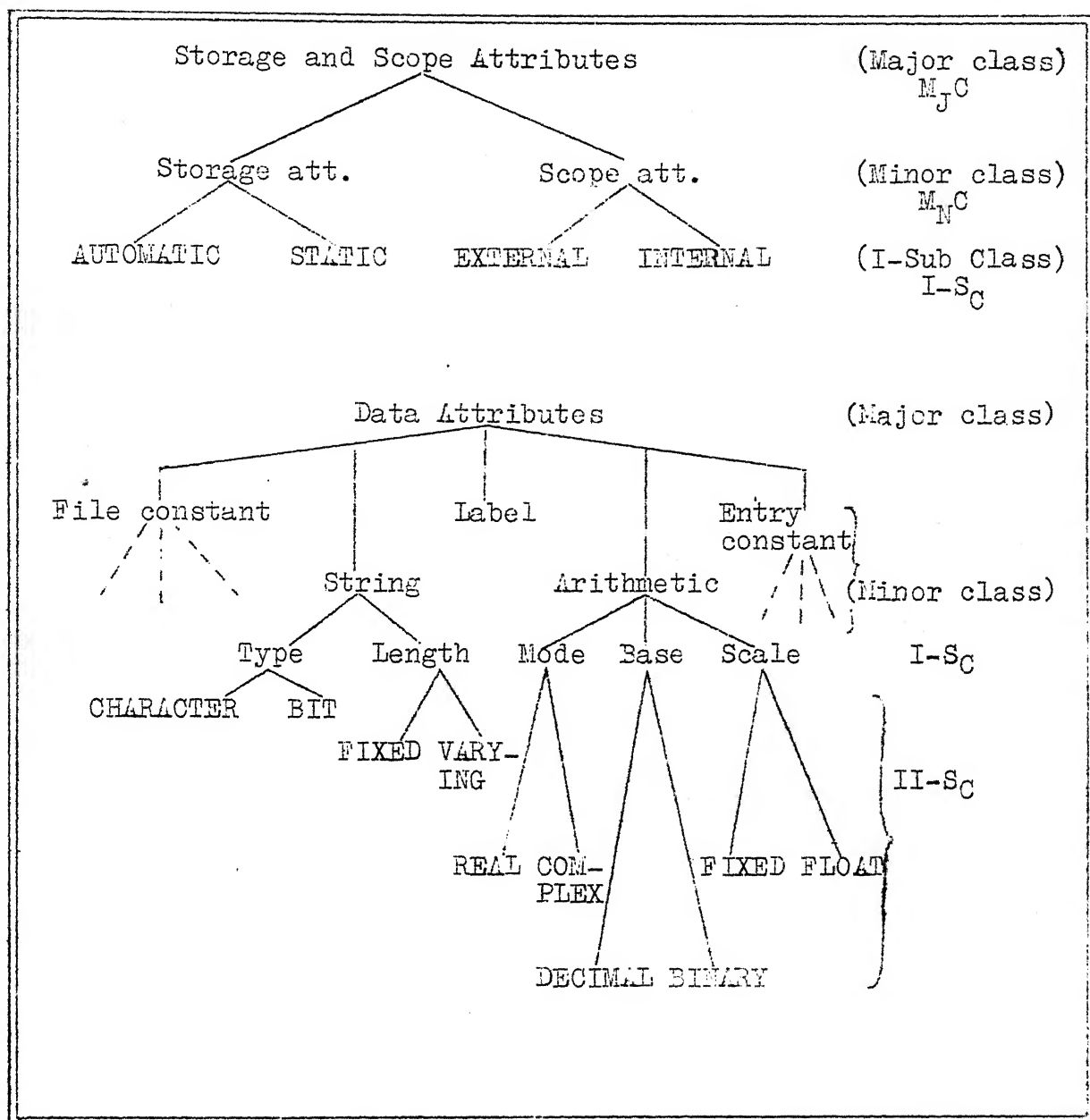


Figure: DC-3.

The character position 'C' represents the major class. The attributes REAL, DECIMAL and COMPLEX have the same number (2) in M_7C since all of them belong to the data attribute class. Similarly STATIC and EXTERNAL have same major class number. Character position 5 represents the minor class. The minor class number of REAL, DECIMAL and COMPLEX is same since all of them belong the arithmetic attributes subclass of the major class-data attributes. Minor class number of STATIC and EXTERNAL is not same since they belong to the storage class and the scope class respectively, both of which are minor classes of the major class - storage and scope attributes. Character position 3 represents the second sub class. For all the four major classes the division may not go upto the second sub class. The remaining characters are used for other purposes which are not logically important.

The numbers assigned to various classes and sub classes are in powers of 2. In other words each number corresponds to one particular bit in the computer word. This sort of assignment helps in collection and classification of attributes. For these purposes the logical operations AND and OR are used, each of which takes only 2 machine cycles to get executed.

The collection routine first determines the major class in which the attribute lies. Depending upon the major class, further actions are taken. For example, for attributes of

data class, it checks that all the attributes declared for a symbol belong to the same minor class. To check this the representative word for data major class, is logically ANDed with the complement of the attribute code word. If fifth character is not zero, it indicates error. This way illegal declarations like 'LABEL DECIMAL' are detected and error condition is raised.

Now it repeats the same process for the first sub class to see that no conflicting declarations are made. For this the RW of data type is logically ANDed with the attribute code word and if 4th character position is not zero, it indicates the error of the type 'REAL COMPLEX'. When validity of the attribute use is checked, a logical OR operation is performed between the corresponding RW and the attribute code word. This operation enters the information about the present attribute in the RW. Consider the case of 'REAL DECIMAL'.

After REAL has been entered in the RW, the RW would be as follows,

		0	0	0	1	1
--	--	---	---	---	---	---

Once DECIMAL is also entered in the RW, it would appear as

		0	0	2	3	1
--	--	---	---	---	---	---

The bit pattern would be as follows

The bit pattern would be as follows:

	000000	000001	000001	REAL
	000010	000010	000001	DECIMAL
	000010	000011	000001	REAL DECIMAL

At the end of collection phase, '1' in the 5th character position of 'data--RW' will indicate arithmetic minor class, '3' in character position 4, will indicate that both mode and base are specified. However, to know exactly which of the four combinations,



is specified, character position 3 would be used. This way the the job of the next phase becomes quite straight forward.

During the collection phase itself, some of the attributes are analysed also. (However some of these attributes can not occur with factorisation). Attributes LIKE, INITIAL, MAP etc. come in this category. After the analysis of these attributes is over control may once again come to the collection phase (for MAP) or it may go to the next phase (in INITIAL, for example).

There are a number of attributes, like dimension attribute, string length attribute etc., which occur inside a pair of parentheses. These attributes need not always be specified. The exact meaning of an attribute in parentheses depends upon the preceding attribute or lexunit in general. For example if '(10)' comes after a symbol, it represents dimension attribute, whereas if it comes after the attribute CHARACTER, it represents string length. Each of these attributes is analysed by a different routine. If the job of determining whether a left parenthesis occurs after an attribute is left to the individual routine, then for each attribute this check would have to be carried out which would make the collection phase considerably slower. The effect of increase in time may not be that pronounced in non factored case, but definitely the increase will be quite marked in factored case where the list of declared attributes is scanned twice.

The scheme adopted for this purpose avoids the individual checking and centralises the job of finding out the context of the LP use. For this purpose each of the attributes is given a LP-type number which is zero if LP can not occur after that attribute. When an attribute is obtained, the corresponding type number is placed in a word called LPCODE. Now if a LP occurs and LPCODE is zero, it indicates illegal use of LP and error is given. On the other hand, if LPCODE

is non zero, appropriate routine is called, depending upon the number stored in LPCODE. The list of LPCODE number and the attribute represented by it is given in the adjoining flow chart.

These LP-attribute analysis routines distinguish the calls between non-factored and factored (post RP processing) cases. In the first case the information extracted from the attribute is filled directly in the representative words whereas for the second type of call it is temporarily kept with the routines. When the 'stacked attributes' are analysed, this information is copied from the temporary storage into the representative words. (For 'stacked attributes' refer to the following sections).

Factorisation of Attributes:

For taking care of factorisation in a DECLARE statement, A/R maintains a stack viz 'factor-stack'. It consists of a chained list of cells taken from the free list. Factor stack has three types of cells:

1. Active LP Cells:

They correspond to those left parentheses for which matching right parentheses are yet to be found. They are needed for matching the right parentheses found with the corresponding left parentheses (to decide the range of factorisation).

2. Inactive LP Cells:

They correspond to those left parentheses for which matching right parentheses have already been found. In fact these cells could have been removed from the factor stack, but it requires breaking a chain from in between and then making it again. This process is quite time consuming as well as requires extra code hence it is not followed. Instead the type of the cell is changed so that same branching mechanism, as for active LP cells and symbol or identifier cells, is made use of for skipping the inactive cells.

3. Identifier Cells:

These cells correspond to the identifiers declared inside the factorisation.

The first word of a cell (for each of the three types) is used for establishing the backward link. The remaining five words correspond to the five representative words. This way information about the attributes declared is carried along with the symbol.

When a call is made to FCSTRC/FCORDN, a fresh cell is taken from the free list and is added on top of the factor stack. In this mode at the end of the collection phase, the representative words are copied into the added cell. The execution of the remaining phases is skipped and control returns to the calling routine.

In post RP attribute analysis mode (RPSTRC/RPORDN), the attributes declared apply to all the names that fall in the range of this right parenthesis (i.e. upto the first active LP in the factor stack). To make these attribute available at the time of the collection phase for each of these name-cells, another stack is needed. It is for this reason that AAR maintains another stack, called 'attribute stack'.

In this mode the collection phase itself works in two distinct phases. In the first phase, all the attributes declared are stacked on the attribute stack, LP-attributes are analysed and LP is stacked on the attribute stack. After this phase is over, this attribute stack is used in the second phase.

In the second phase, next cell in the 'factor-stack' is taken, its contents are copied into the representative words and the attribute stack pointer is initialised to the bottom most attribute. Now stacked attributes are taken one by one and given to the non factored collection module, which analyses them as usual. Bringing the contents of the factor stack identifier cell into the representative words takes care of the attributes declared previously. This way factored declaration environment is transformed to non factored declaration environment.

At the end of the collection phase for this identifier, if PRNCNT is not zero i.e. factorisation is still incomplete, the representative words are copied back into the old cell of the factor-stack. However, if factorisation is complete i.e. PRNCNT is zero, this cell is given to the remaining phases. These phases after completing their job send the control back to the collection phase. This process is repeated for all the identifier cells of the factor stack till an active LP cell is obtained, showing the end of the range of the present factorisation. If PRNCNT happens to be zero all the cells of the factor stack are returned to the free list. However, if PRNCNT is not zero, the active LP cell is converted into an inactive LP cell. Finally control is returned to the declare statement routine.

3.4.2 Making the Complete Attribute Set:

Attributes collected during the collection phase are to be analysed to see whether the attributes defined form a complete set or some attributes are to be supplied to complete the set. Consider an example in which case only REAL EXTERNAL are defined as attributes. Since no storage class attribute is specified, it would have to be supplied to complete the storage and scope attributes. In this case STATIC attribute would be supplied by the compiler. Similarly the only data

class attribute specified is REAL which represents the arithmetic minor class. The sub class is 'mode'. The two remaining sub class attributes 'base' and 'scale' would have to be supplied. Completed set would be EXTERNAL STATIC REAL DECIMAL FLOAT. (The completed data class corresponds to the 'Standard default actions of PL/I'. However PL-7044 would complete it as REAL-floating point or REAL-integer depending upon the first character of the name). In case no data class attributes are specified, default attributes are assigned depending upon the first character of the symbol. Similarly if no storage and scope attributes are defined, default storage and scope attributes are assigned. Details of default actions are given in Appendix-G.

For a minor structure, if no attributes are specified, the control is returned to the DECLARE-routine. In case the DECLARE statement realises that the symbol should have been assigned attributes (after seeing the level number of next element), it calls the default attribute assignment routine.

3.4.3 Storage Assignment and Output Generation:

Every declared variable is to be assigned storage area. Exact amount of storage area depends upon the attributes of the variable. Storage area assigned depends upon the storage class attribute also. For arrays or character scalars, storage is assigned from the run time stack. For doing this, at run time,

output code is to be generated. For this purpose the variables are classified under six categories.

Static : Scalar and Array

Normal : Scalar and Array

Formal : Scalar and Array

For the purpose of storage assignment the static variables are treated as being declared in the block '0'. This assumption, however, does not change the declared scope of that variable. This ensures that storage space assigned to these variables would not be released throughout the execution of the program, since block '0' would never be closed. The salient advantage of this scheme is that all the routines doing array allocation and initialisation, scalar allocation and initialisation etc. for AUTOMATIC variables can also be used by the STATIC variables.

As specified in the PL/I specifications, all the storage assignment and initialisation for STATIC variables should be done only once in the beginning, irrespective of the number of times the block in which the declaration has been made is invoked. To conform to this specification all the output generated for STATIC variables is assembled under a different location counter called 'STATIC'. This counter gets priority over the normal counter viz. 'CODE'. The control is transferred to start with STATIC variables storage assignment and initialisations. It is because of this reason no expression is allowed

in array sizes, array bounds, string lengths, iteration factor etc. for a STATIC variable.

For normal variables output generated, if any, goes under the location counter 'CODE'.

For formal parameters, prologue list word(s) is(are) to be generated. The prologue list word carries the information about minor type of the variable, and address of the storage assigned to this formal parameter. These words are assembled under a location counter called 'PROLOG', which arranges all the prologue list words in a sequence. The PROLOG counter becomes necessary because of two reasons. Firstly the normal declarations and formal declarations can be inter-mixed and secondly, explicit declaration of a formal parameter need not be given by a DECLARE statement. Since it can only be recognised when all the DECLARE statements corresponding to that block are processed, PROLOG counter becomes a necessity. Prologue list interpretation routine written, expects the prologue list in consecutive words. For details about prologue list, reference should be made to Chapter PC.

For all the variables, storage address is assigned in terms of the offset from the start of the storage area for that particular block. The offset count is updated by the attribute routine if any storage assignment is done.

The storage assignment module maintains a table, called EXTERNAL-table, of all the definitions of variables having EXTERNAL scope. When a symbol gets declared with EXTERNAL attribute, the module first checks whether that variable exists in the EXTERNAL-table or not. If an entry with same name already exists in the EXTERN-table, its major type and minor type are compared with the major type and minor type of the present definition to see whether they are consistent or not. If these do not match, error condition is raised, since both the definitions represent the same variable. If types are found to be consistent, reference to this definition is directed to the old definition. However, if a definition doesn't exist in the EXTERN-table, storage is assigned as usual and an entry is made in the EXTERNAL table.

3.4.4 Filling the Symbol Table Entries:

Entries like major type, minor type, precision etc. are stored in the symbol table. Control then returns to the calling routine.

If a call comes for default attribute assignment, the first phase is skipped and control directly goes to the second phase.

3.5 Analysis of Initial Attribute:

Initial attribute requires maximum amount of code generation and is quite interesting also. Its implementation details would be described here.

The initial attribute specifies an initial constant value to be assigned to a data item when storage is allocated to it.

3.5.1 General Format:

INITIAL (initial item [, initial item]...)
where "initial item" is

or [(i-f)] extended string constant
or [(i-f)] [+|-] arithmetic constant
or [(i-f)] *
or (i-f)(initial item [, initial item]...)

where,

"i-f" is an "iteration factor" which can be a scalar expression. The scalar expression is evaluated at the time of initialisation and is converted to integer value.

"extended string constant" is

[(r-f)] string constant

where "r-f" is repetition factor.

"arithmetic constant" is

Complex constant or

Real constant

where "complex constant" is

[[+|-] real-constant] $\left\{ \begin{matrix} + \\ - \end{matrix} \right\}$ imaginary-constant

where "real constant" is

integer constant or

floating point constant , and

"imaginary-constant" is
real-constant I

3.5.2 General Rules:

1. The INITIAL attribute can not be given to major structure names, minor structure names, and parameters.
2. For "Static Variables" iteration factor should be optionally signed constant only.
3. Only one scalar value can be specified for an element variable; more than one may be specified for an array variable. A structure can be initialised only by separate initialisations of its elementary names.
4. Scalar values specified for an array are assigned to successive elements of the array in row-major order.
5. If the number of values specified for an array exceeds the number of elements of the array, excess values are ignored.
6. If the number of values assigned is less than the number of elements in the array, the remainder of the array is not initialised.
7. 'Asterisk' specifies that corresponding position is to be left un-initialised.
8. If the iteration factor is followed by a constant, it specifies the number of consecutive array elements that are to be initialised with the given value. If the iteration factor is followed by an asterisk, it specifies the number of consecutive array elements that are to be skipped in the initialising operation. If the

iteration-factor is followed by a list of initial items, then the list is to be repeated the specified number of times with each item initialising the specified number of array elements.

9. Only 10 levels of iteration factor nesting is allowed in initial item by PL-7044 Compiler.
10. A negative or zero value for the iteration factor causes no initialisation.
11. Only label constants can be given as initial values for label variables, the name of such label constant must be known within the block in which the label variable declaration occurs.
12. An alternative method of initialisation is available for elements of array of non-STATIC statement label-variables. An element of a label array can appear as a statement prefix, provided that all subscripts are (optionally signed) integer constants. The effect of this appearance is the initialisation of that array element to a value that is a constructed label constant for the statement prefixed with the subscript reference. This statement must be internal to the block containing the declaration of the array. Only one form of initialisation can be used for a given label array.

13. INITIAL attribute should be the last attribute in the list of attributes. e.g. REAL INITIAL(0) is all right but not INITIAL (0) REAL.

INITIAL attribute can be factored, but it should be at the factoring level of zero i.e. when "PRNCNT" is zero. These restrictions are not part of PL/I and they belong to PL-7044 only.

3.5.3 Implementation:

(i) Examples:

Implementation of INITIAL attribute would be explained with the help of the following examples:

- E-1. DECLARE (A FIXED, B STATIC, (C,D) COMPLEX) INITIAL(-1);
 E-2. DECLARE (CH CHARACTER, BT BIT) INITIAL('101');
 E-3. DECLARE (INT, JINT, REL, CMP COMPLEX, BIT BIT)(10,5)
 INITIAL (1,(5)*,(K*J/L)((3)2.0,3+1.I),*,'101'B);
 E-4. DECLARE LAB(10) LABEL (L1, L2, 3(L3,L4),*,L5);

As can be seen from here, the same INITIAL-list is being used for initialising variables of different types viz. in E-1, integer, floating point and complex scalars are sharing the same initial list. To do the right assignment, conversions are required. Conversions can either be done at compile time or at run time. If conversions are postponed for run time, lot of time would be wasted since every time a block is invoked, initialisation would have to be done afresh. In case of factored declaration use of INITIAL

attribute would be still more inefficient because if two variables of same type share a list which requires conversion, it would have to be done for both of the variables. Then situation becomes still worse if a conversion is involved with iteration factor since then it would have to be repeated that many times even in one initialisation. Thus there is overwhelming incentive in doing conversions at compile-time itself. It is for this reason that PL-7044 forces that INITIAL be the last attribute so that by the time INITIAL attribute is obtained all information is available about the type of the variable. This makes possible for two or more variables to share the same initial-list with no loss in execution time.

Treatment given to the scalar initialisation and array initialisation is different and hence they would be described separately.

(ii) Scalar Initialisation:

For scalar initialisation the code to do the initialisation is generated in open ended manner and no special routine is called for doing initialisation. However, in case of string variables and label variables (with list), assignment routines are called to do the initialisation since they require more work than just assignment. The constants are converted to match the type of the variable, if possible, and they are outputted in second pass itself. The code for doing the initialisation is

generated in third pass. Constants, therefore, appear out of the sequence of the normal code.

For E-1, the constant area and the code area would be as follows:

Constant Area (Outputted in Second Pass):

```

      USE  JUNK
LO0001 EQU  =0-1           integer '-1'
LO0002 EQU  =06014000000000 f.p. '-1.'
LO0003 EQU  *              Complex '-(1.+0I)'
      OCT  6014000000000    real part '-1.'
      OCT  4000000000000    imaginary part '-0'

```

Code Area (Outputted in Third Pass):

```

      USE  CODE
      CLA  LO0003
      STO  add D
      CLA  LO0003+1
      STO  add D+1
      CLA  LO0003
      STO  add 3
      CLA  LO0003+1
      STO  add C+1
      USE  STATIC
      CLA  LO0002
      STO  add B
      USE  CODE
      CLA  LO0001
      STO  add A

```

It should be noted that:

- i) Constants are outputted under a location counter called JUNK and the initialisation code for AUTOMATIC variables, e.g. A, C and D, is outputted under the location counter 'CODE' whereas the code for initialising a STATIC variable is outputted under 'STATIC' location counter.
- ii) "add X" represents the address of the variable X, if it is of one word type (integer or floating point), otherwise it represents the address of the first of the series of consecutive words assigned to that variable. It is of the type 'BS.bn+offset' where "bn" is the block no. and "offset" is the offset of this variable from the starting of the storage assigned to this block i.e. 'BS.bn'.
- iii) For one word constants, literal management is left to the MAP assembler (by generating 'EQU' code) whereas for multiword assignment, literal management becomes the responsibility of the compiler itself. It would be proper, if only one copy of the multiword literal exists in the code. However, this sort of optimisation has not been tried in this compiler.
- iv) Variables of same 'minor-type', (sharing the same initial list) share the same constant at run time also. This is evident in case of complex variables C and D.
- v) There is scope for further optimisation. It can be seen from the example given, that the complex constant

is brought into the accumulator twice, once for initialising C and second time for initialising D. It could have been so organised that in case of factored initialisation a constant is brought into the accumulator only once. This was, however, not tried.

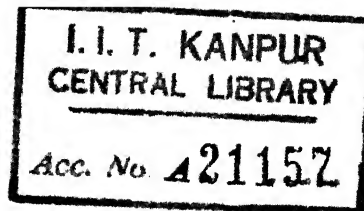
As already mentioned, for string variables run time assignment routines are to be called. The code generated for E-2 would be as follows:

Constant area:

USE	JUNK	
L00001 EQU *		
OCT	5000000000000	bit string '101' left justified
OCT	3	length of bit string
L00002 EQU *		
OCT	3	length of character string
BCI	1,101	Actual character string

Code Area:

USE	CODE	
TSX	BITASN,4	BIT Assignment routine
PZE	add BT	address of variable BT
PZE	L00001	Bit string constant address
TSX	CHRASN,4	Character assignment routine
PZE	add CH	Address of variable CH
PZE	L00002	Character string constant address.



It would have been noted in both the examples that the sequence of code generated is exactly the reverse of the physical appearance of the variables in the DECLARE statement. This is because of the mechanism employed to analyse the factored declaration, described previously.

c) Array Initialisation:

Unlike scalar initialisation, array initialisation is done by calling a routine, specially coded for this purpose, since array initialisation is more involved because of iteration factors, use of asterisks etc. The calling sequence of the array initialisation routine is as follows:

```
TSX  ARINTL,4
PZE  header,, mnrtyp
PZE  intlst
```

where,

"header" is the first order storage address of the array header, "mnrtyp" is one of the 12 minor types (Appendix-D) and "intlst" is the address of the run-time list generated for the list specified in the DECLARE statement. As in scalar initialisation, repetition of a constant is avoided; here also repetition of initial list is avoided. Two or more variables having same type share the same initial list.

Format of Initial List:

Initial list is an ordered collection of eleven different types of elements. Each element consists of one or more words.

First word identifies the type of the element and the remaining words, if any, store the constants etc. The routine ARITHL interprets this list and recognises the type of the element with the help of the identifying word. The types of the elements and their word requirements are as follows:

<u>Type Number</u>	<u>No. of words required</u>	<u>Remarks</u>
0	1	End of Initial List
1	2	Value of constant iteration factor in the second word
2	2	Value of the expression in iteration factor in the second word
3	2	Integer constant in the second word.
4	2	Floating Point constant in the second word
5	3	Bit string in the 2nd word and its length in the third word
6	2+n	Character string length in 2nd word and actual character string in next n consecutive words (n depends upon the number of characters).
7	3	Complex constant. Real part in the 2nd word and imaginary part in the 3rd word.
8	2	Label scalar constant in the second word
9	1	Asterisk
10	1	End of iteration factor range

ARINTL routine calls another run time routine F.RNGE which supplies the first element address, last element address and no. of words per element of array. These three quantities are used to get the next element address of the array and to decide when the array is over. This routine maintains a ten word stack where it pushes the address of the iteration factor in initial list, and also the iteration count. The iteration count is decreased by one when it meets the end of iteration factor range. Now it again starts initialising further elements from the same initial list portion, address of which is available on top of the stack. This process is repeated till iteration count is reduced to zero, then the top most cell is popped off from the stack and next element is taken from the initial list.

The routine ARINTL returns control when either the end of initial list code is met in the initial list or all the elements of the array get exhausted.

Corresponding to the example E-3, four initial lists would be generated. Variables INT and JINT, both being integer, would be sharing the same initial list, generated with all constants converted to integer type. Floating point variable REL, complex variable CMP and bit string variable BIT would be having three different initial lists of types floating point, complex and bit string respectively.

The initial list generated for integer variables INT and JINT would be as follows:

	USE	JUNK	
L00001	EQU	*	
	PZE	3	Integer constant -
	OCT	1	1
	PZE	1	Iteration factor (constant) -
	OCT	5	5
	PZE	9	'*'
	PZE	10	End of iteration factor range
	PZE	2	Expression in iteration factor -
L00002	PZE	**	Value of K*J/L would be filled
*			here
	PZE	1	Constant iteration factor -
	OCT	3	3
	PZE	3	Integer constant -
	OCT	2	2.0 Converted to 2
	PZE	10	End of Iteration factor range
	PZE	3	Integer constant -
	OCT	3	3+1.I converted to 3
	PZE	10	End of Iteration factor range
	PZE	9	'*'
	PZE	3	Integer constant -
	OCT	5	'101' B converted to 5
	PZE	0	End of initial list

It is to be noted that:

- (i) Initial list address is L00001,
- (ii) Code for evaluating the expression K*J/L would be outputted by the expression processor. The value, converted to integer if necessary, would be filled by the same routine in the address portion of L00003. The word L00003 is given from the code under a different location counter, called 'BSS'. It becomes necessary to do so because the value is needed by more than one list, in general. Before generating the code for calling ARINTL, code for copying the value from L00003 into L00002 would have to be generated. For non-factored case this extra word is not necessary and value is directly filled in the initial list.

Use of a function inside iteration factor is not prohibited. However, a recursive call can not be given to the same procedure because we are storing the value of the expression in code itself, old value would be lost if a second call is made to the same procedure. It could have been taken care of by providing a temporary location from the temporary area (which is saved when a recursive call is made to the same procedure) and then taking the value from this temporary every time it is needed. However, it was not tried because it would slow down the execution for the non-recursive case also.

(d) Label Variable Initialisation:

Label variable initialisation as given in example E-4 is similar to that of any other initialisation. The only difference being that in place of ordinary constants, label constants take part in it.

There is another way of initialising the elements of arrays of label variable by prefixing it to any statement except PROCEDURE, ENTRY, DECLARE, FORMAT, ELSE statements.

Consider the trivial program:

```

MAIN: PROCEDURE OPTIONS (MAIN);
      DECLARE L(10) LABEL;
      GOTO L (1);
L(1): PUT LIST ('LABEL INITIALISATION DONE');
      STOP;
      END MAIN;

```

Execution of this program should print the following text:

LABEL INITIALISATION DONE.

To take care of this type of label initialisation, following scheme is proposed.

A Label Initialisation Table (LIT) is maintained in second pass. LIT has one word for each of the blocks defined in the program. These words are initialised to zero to start with. A chained list of all label initialisations of label prefix type for one block is outputted in second pass under the location counter BSS. The starting address of the chain for a block is stored in the label initialisation table. This chained list is interpreted at run time by label initialisation routine, immediately after all the declarations are over for the block being invoked. Label initialisation routine is supplied with the starting address of the chain in a word. If the contents of this word happen to be zero, it indicates that there is no label initialisation, of prefix type, for that particular block.

The list consists of chained cells, each of two words. One cell is outputted for one label initialisation of this type. The format of the two word cell is as follows:

	LSTCELL		HEADER
	LBLCONS		ARGLST

where,

LSTCELL: Last cell address, used for linking the chain.

HEADER: Label array header address.

LBLCONS: Constructed label constant containing word
address

ARGLST: Address of the list of subscripts used in the
label initialisation.

The calling sequence for the label initialisation routine
is as follows:

```
TSX  LBLINT,4  
PZE  wrdadr
```

where,

" wrdadr" is the address of the word containing the
starting address of the chain.

DECLARE STATEMENT Flow Chart

NOMENCLATURE:

1. Lexical Units:

IDENT: Identifiers (non key words) or key words
INT: Integer

2. Routines:

ATANRT: Attribute analysis routines
ATORDN: ATANRT for ordinary scalars
CHKDFT: Check whether last minor structure has been given explicit attributes, if not assign them.
COVER: Find out the covering item.
ENTER: Symbol table 'ENTER' routine.
FCORDN: ATANRT for ordinary scalars (inside factorisation)
FCSTRC: ATANRT for minor structures/base elements (inside factorisation)
LPENT: Enter '(' on top of the 'factor-stack'
MNSTRC: ATANRT for minor structures/base elements
MJSTRC: ATANRT for major structure
RPORDN: ATANRT for post ')' attributes (for ordinary declaration).
RPSTRC: ATANRT for post ')' attributes (for factored structure declaration).

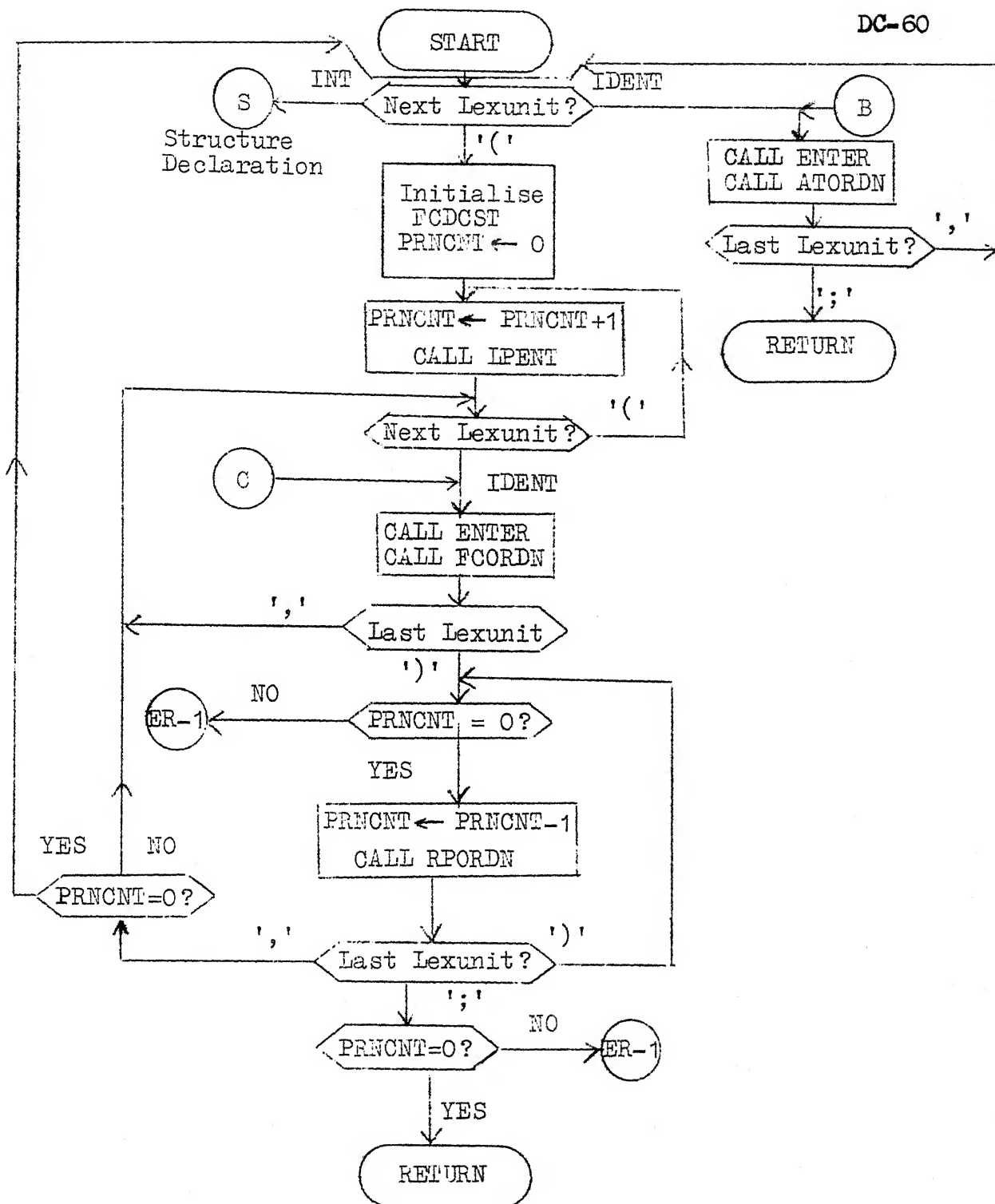
3. Symbols and flags:

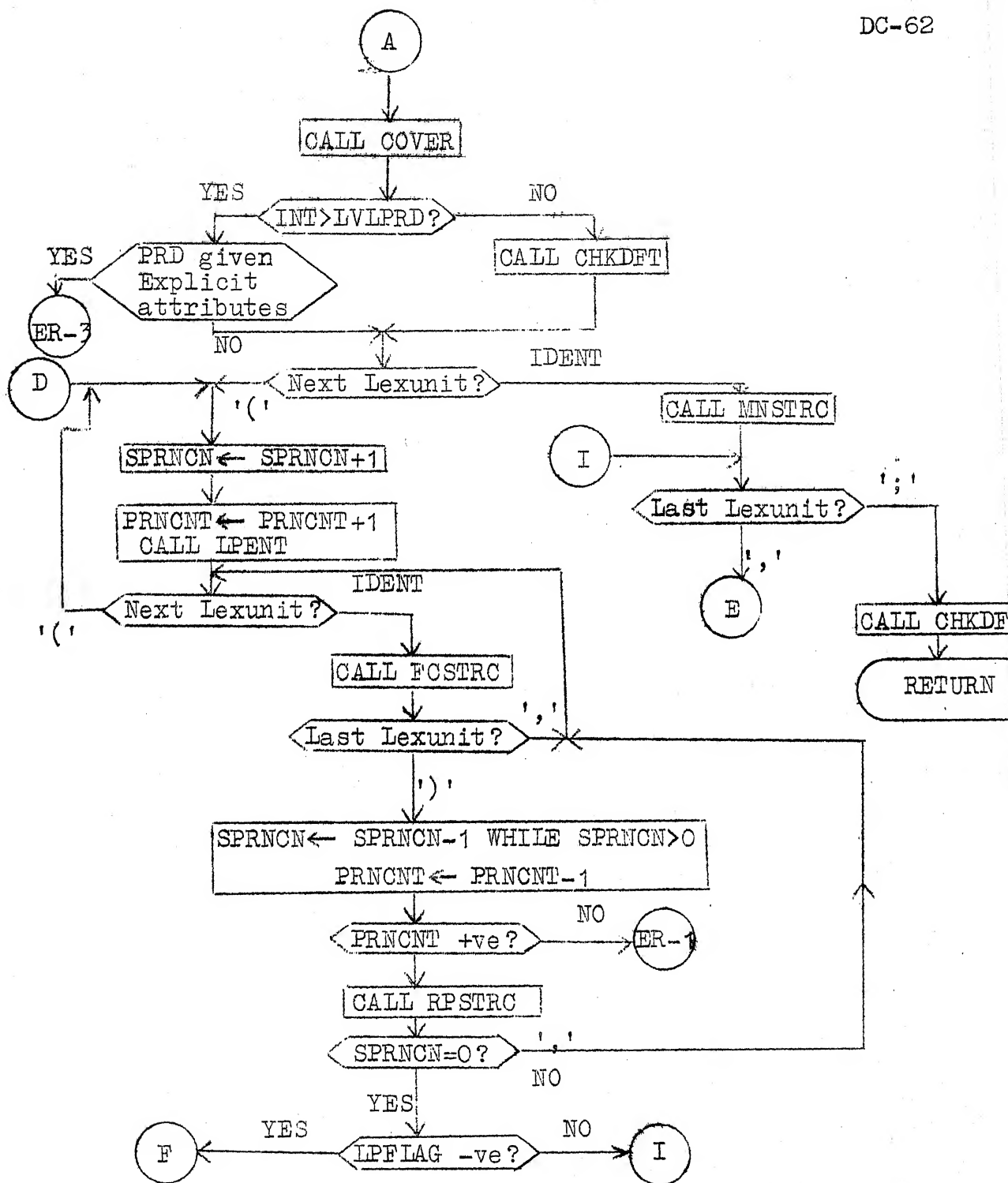
FCDCST: Factor declaration stack
LPFLAG: Flag used for factored structure declaration
LSLVNO: Level number of the last element inside factored structure declaration
LVLPRD: Level number of the last item
PRD: Previous declaration
PRNCNT: Differential count of parentheses
SPRNCN: Differential count of parentheses for level-factored declaration of structure

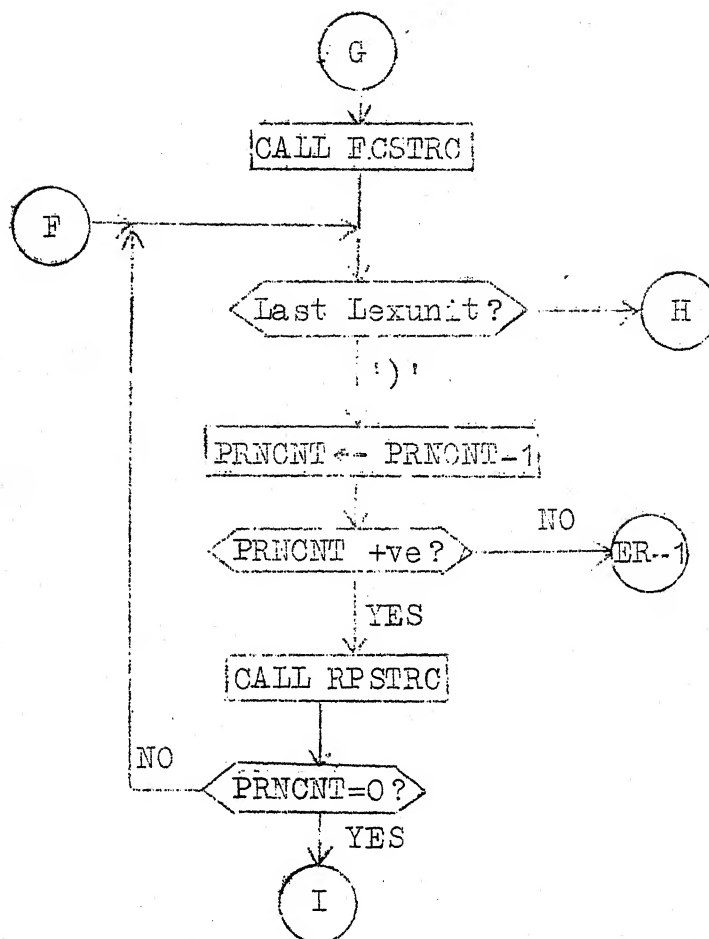
4. Error Exits:

ER-1: Mismatch of left and right parentheses
ER-2: Illegal use of identifier
ER-3: Illegal level number used inside factored structure declaration

NOTE: Lexunits, for which there are no corresponding branches coming out of the decision boxes, constitute error.







(DEC-4)

SYMBOL TABLE Flow Charts

(ENTER and LOOKUP)

NOMENCLATURE:

1. Symbols and flags:

BLKCHR: Character storing the serial number of the block in which the declaration has been made.

COVER: Covering element address of the item presently getting defined.

CVRWD: Word storing the covering element address

LATEST: Contains the address of the added/found cell.

LEVEL: Level number of the item presently getting defined.

LSSNMN: Last in stack with same name

LSSRNM: Last in structure with same name

PSRNO: Serial number of the block open

SRLVCH: Character storing the level number

SRNOCH: Character storing the structure number

STRNO.: Serial number of the structure getting defined presently.

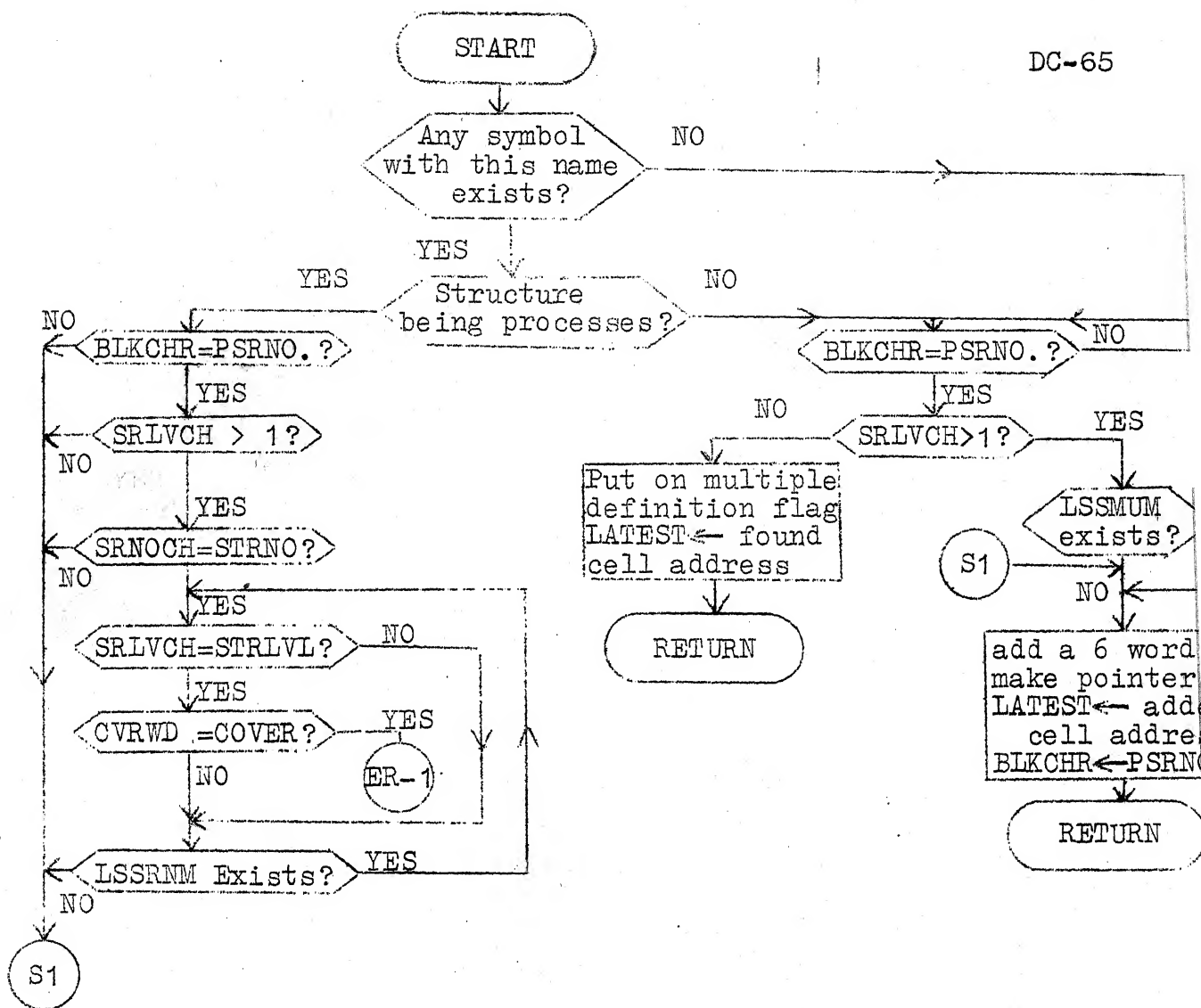
2. Error Exits:

ER-1: Illegal qualification

ER-2: Ambiguous reference

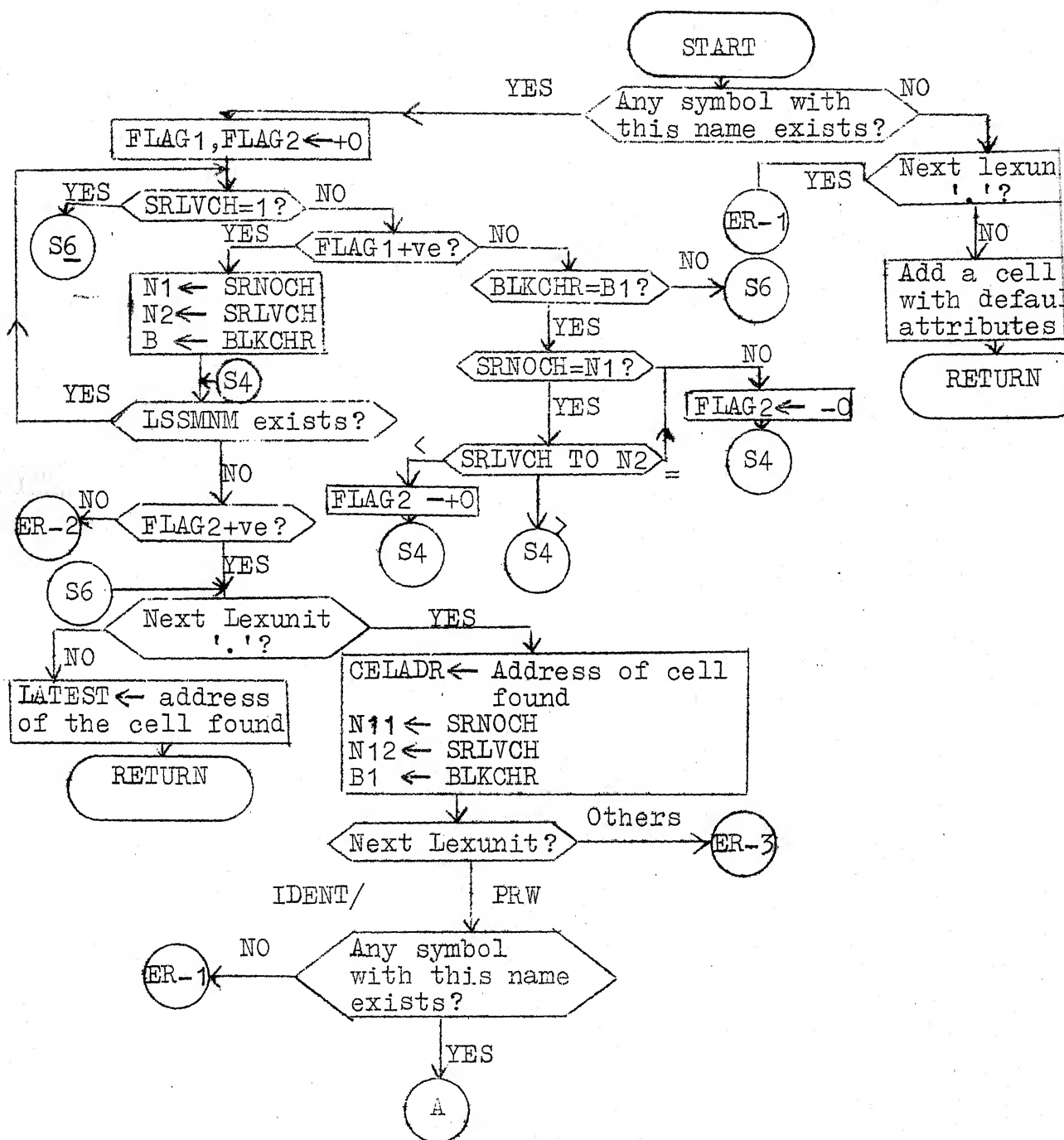
ER-3: Illegal lexunit

ER-4: Ambiguous qualification

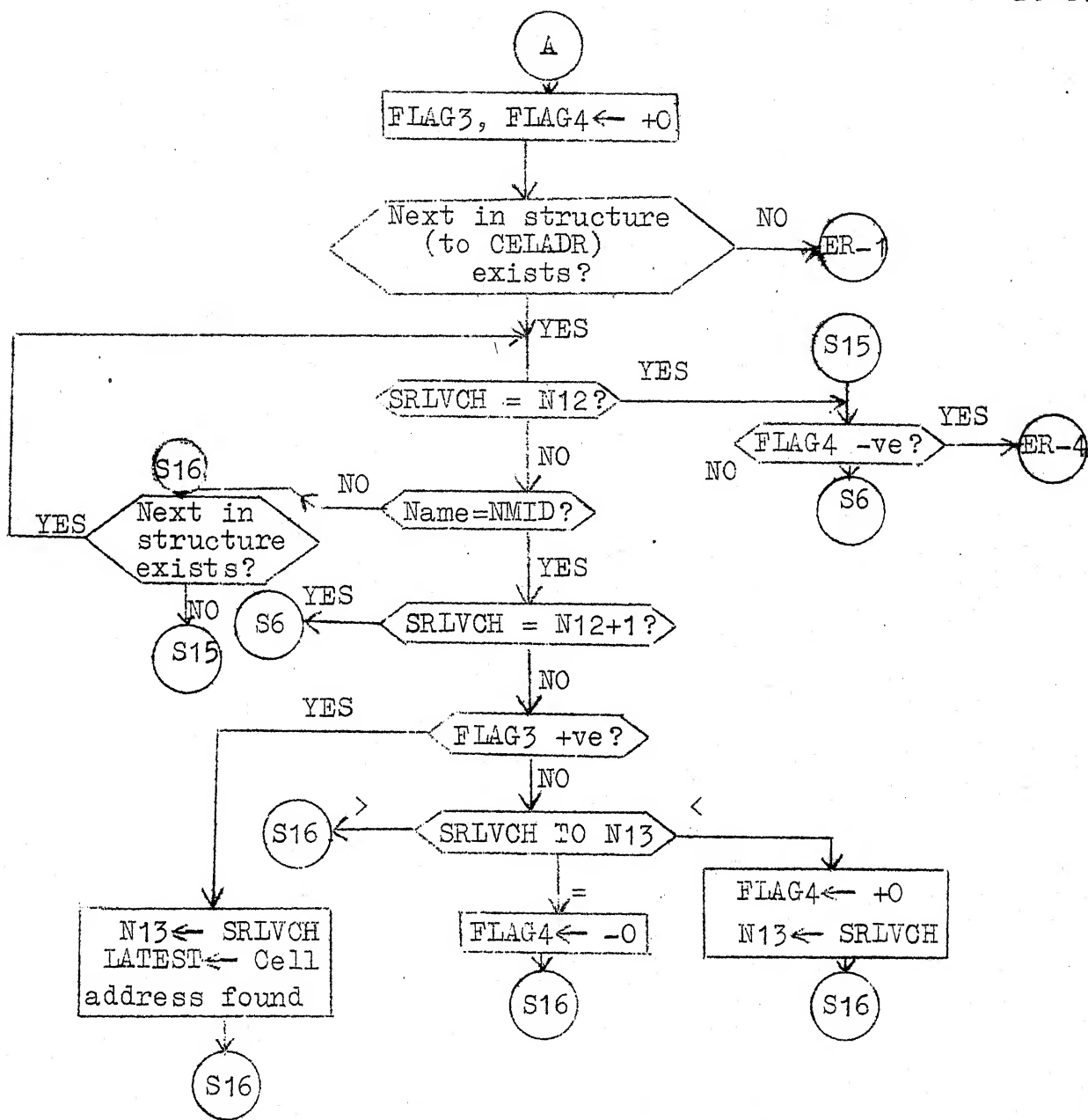


SYMBOL TABLE ENTER

(STE-1)

SYMBOL TABLE LOOKUP

(STL-1)



(STL-2)

ATTRIBUTE ANALYSIS ROUTINES FLOW CHARTNomenclature:

1. Routines:

ADDCEL: Add a six word cell on the top of the 'factor-stack'!!
Copy the five representative words in the added cell.

ASNSTG: Assign storage

ASSIGN: Complete the attribute set, assign storage and fill the entries in the symbol table.

ATBANL: Main attribute analysis routine

CHKFML: Check whether the symbol is a parameter.
If so do the initialisations for outputting the prologue list word.

CHKFRM: Check whether any of the symbols in the factored declaration is a parameter.

COLLECT: Complete the attribute set.

ENTATB: Enter the attribute on the top of the attribute stack.

FACUPD: Analyse the collected attributes.

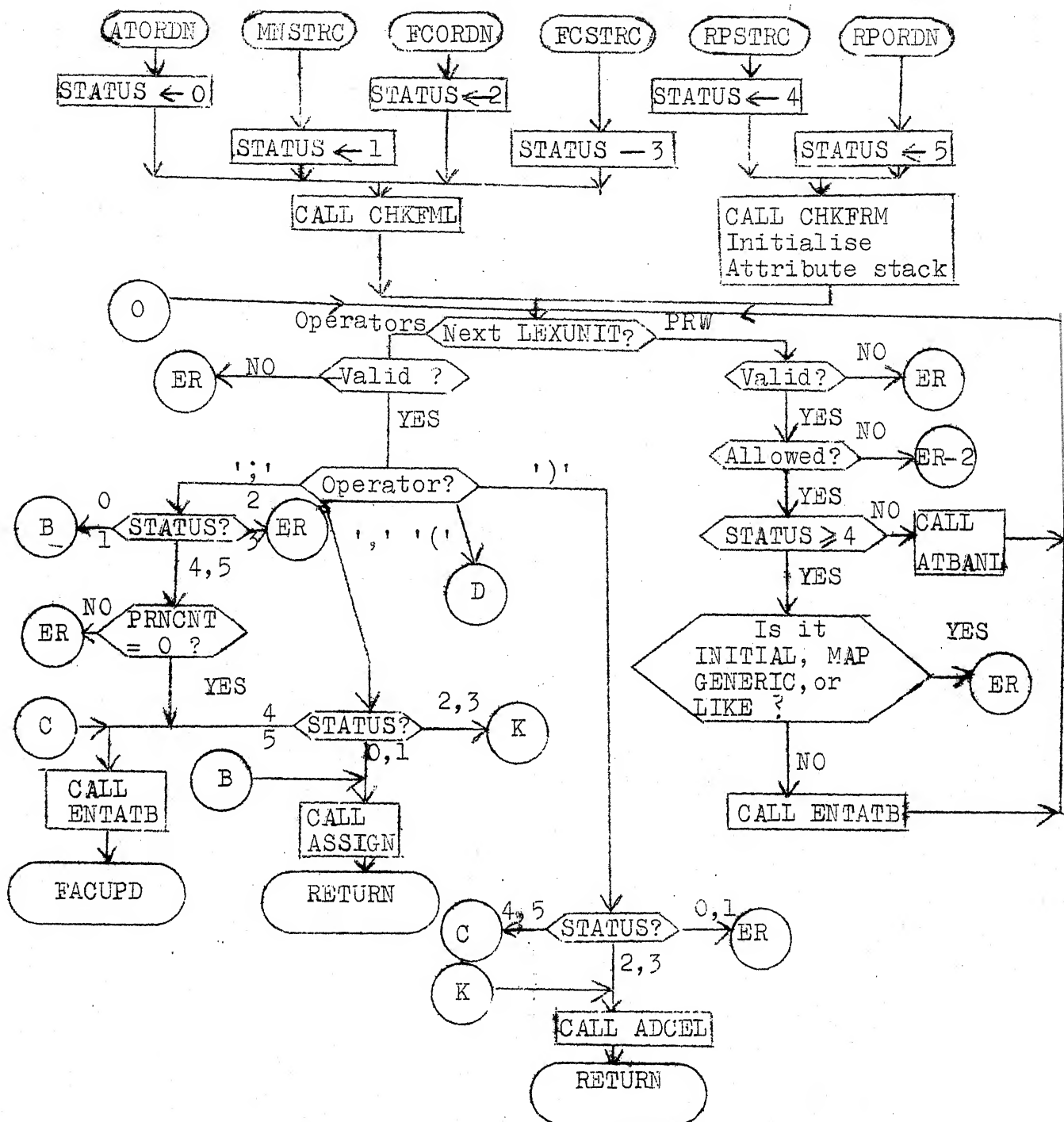
GENERC: Analyse the 'GENERIC' attribute.

GETCEL: Copy the contents of the top most cell of the 'factor-stack' into the five 'representative words'.

INITAL : Analyse the 'INITIAL' attribute
LIKE: Analyse the 'LIKE' attribute
MAP: Generate the 'EXTERN name' card.
RESCEL: Copy back the contents of 'representative
 words' into the top most cell of factor stack.
SYMFIL: Fill the entries in the symbol table.
UPDATE: Update the information about the attributes
 of a sub class obtained till now.
WARNNG: Give warning.

2. Symbols and Flags:

PRNCNT: Differential count of parentheses used in a
 factored declaration.
STATUS: A multivalued switch to determine the type of
 attribute analysis routine to be called.
STGBAN: 'Storage and scope attributes prohibited' flag.
STRUCT: 'Structure declaration' flag.



ATTRIBUTE ANALYSIS ROUTINE

(ATT-1)

D

LP Type No.	ATTRIBUTE
0	Error- Illegal Use of '('
1	SETS/USES
2	Precision
3	RETURNS
4	Label List
5	String Length
6	Entry List
7	Record size
8	Non factored Dimension Attribute
9	Factored Dimension Attribute

Each attribute is analysed by a separate routine which takes care of factored and non- factored declarations both.

O

ASSIGN

START

CALL COLECT
CALL ASNSTG
CALL SYMFIL

RETURN

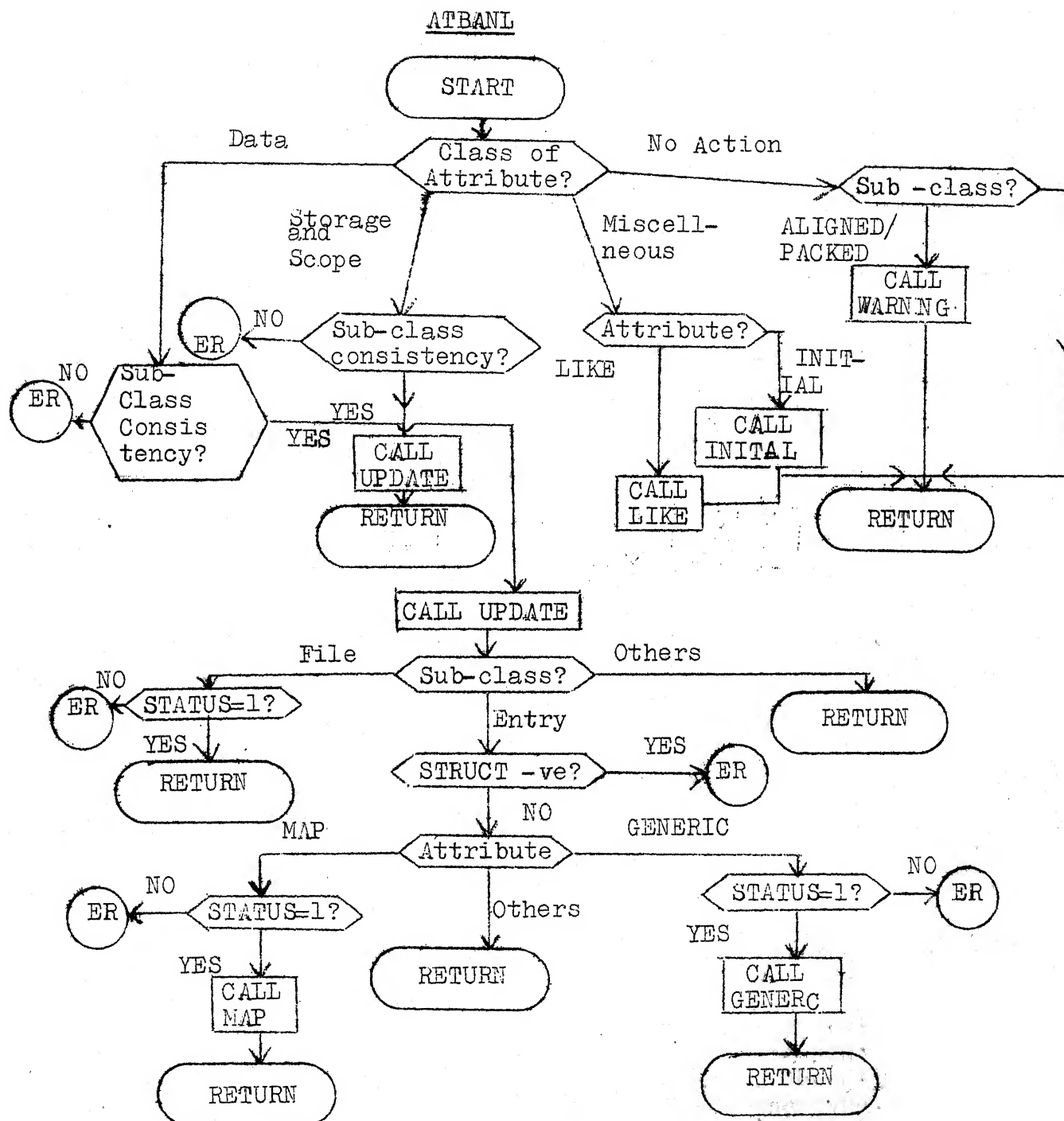
SYMFIL

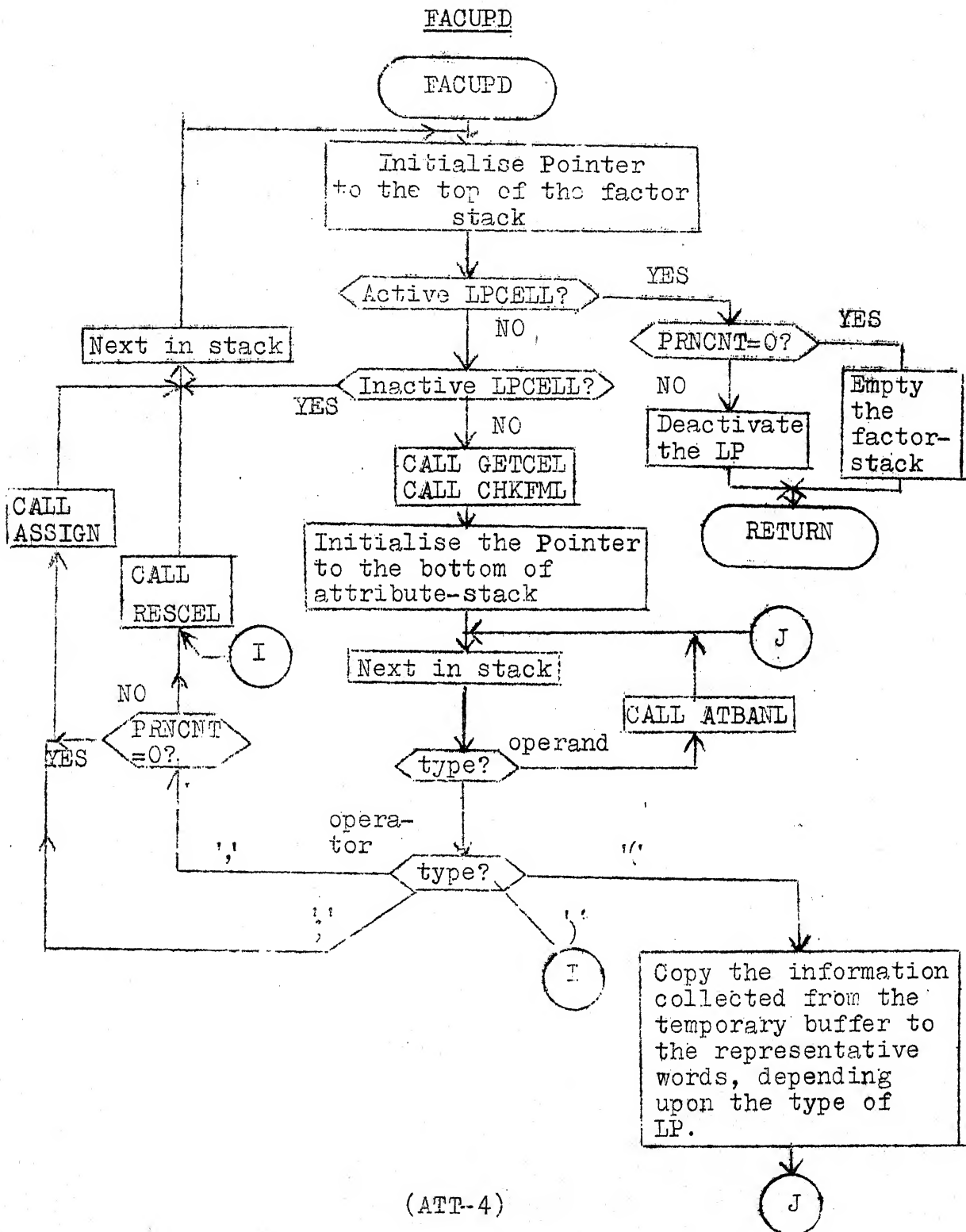
START

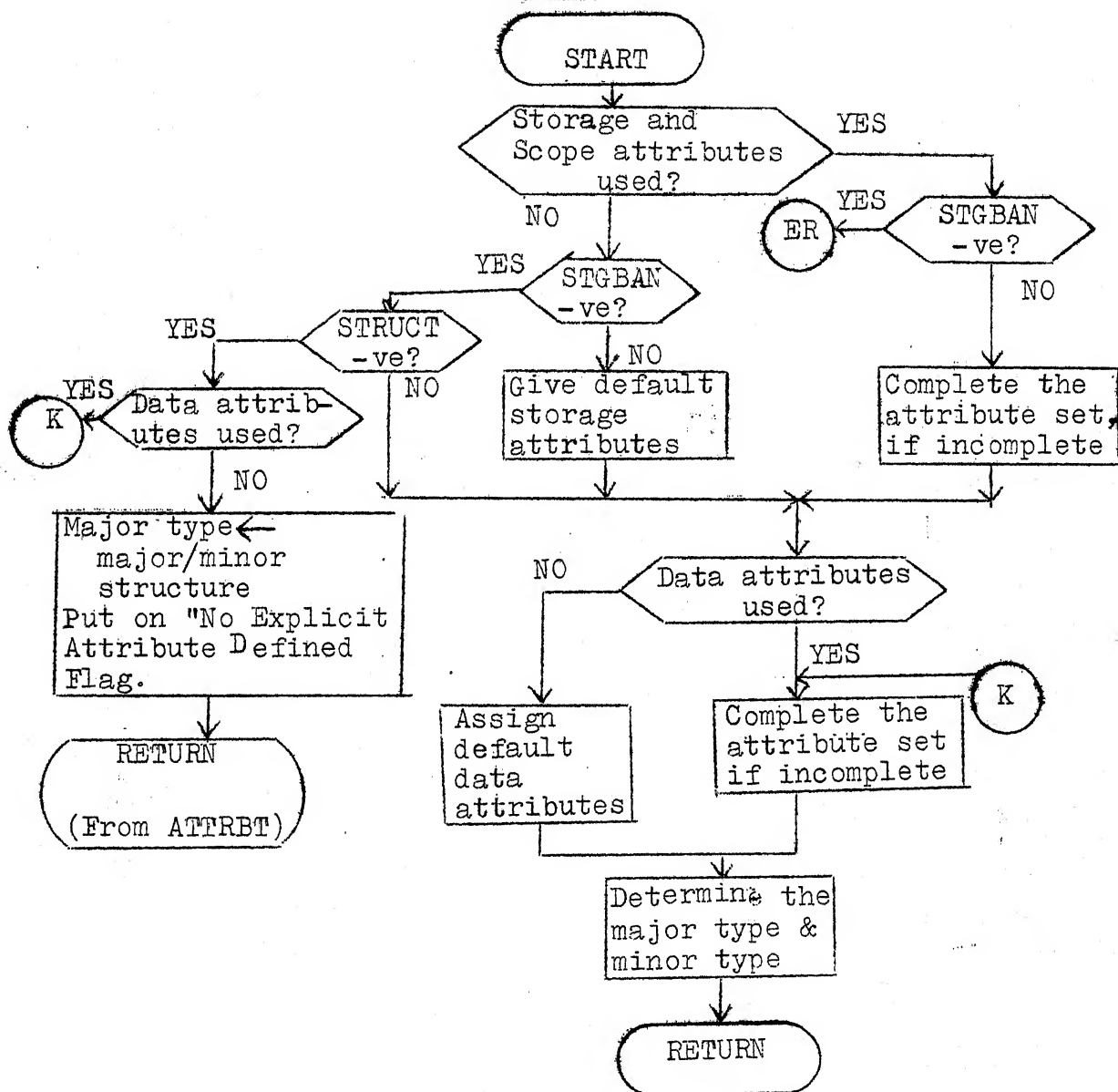
Fill the symbol
table entries with
minor type, major
type, storage
address etc.

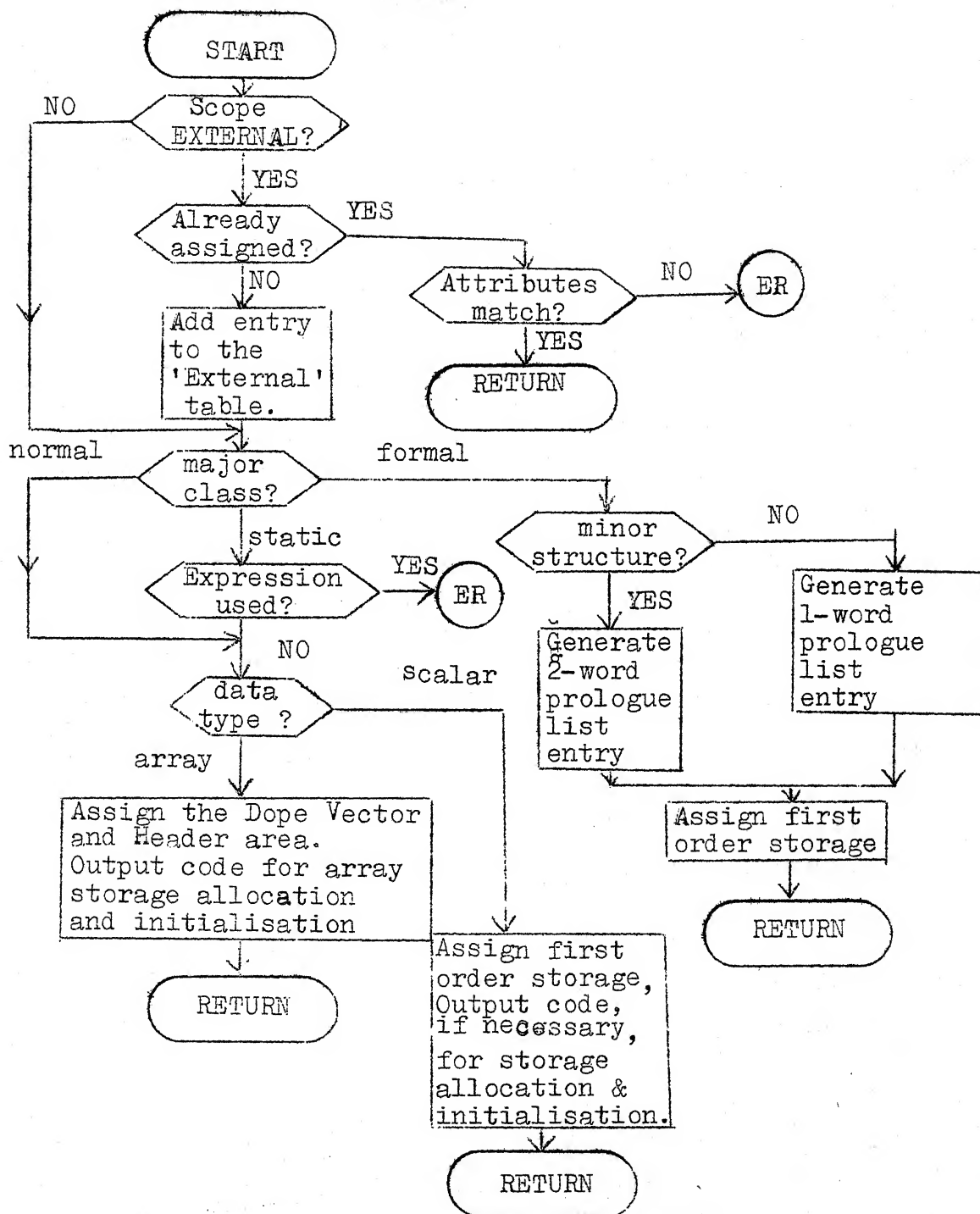
RETURN

(ATT-2)





COLLECT

ASNSTG

'BYNAME' Attribute Analysis Routine Flow Chart

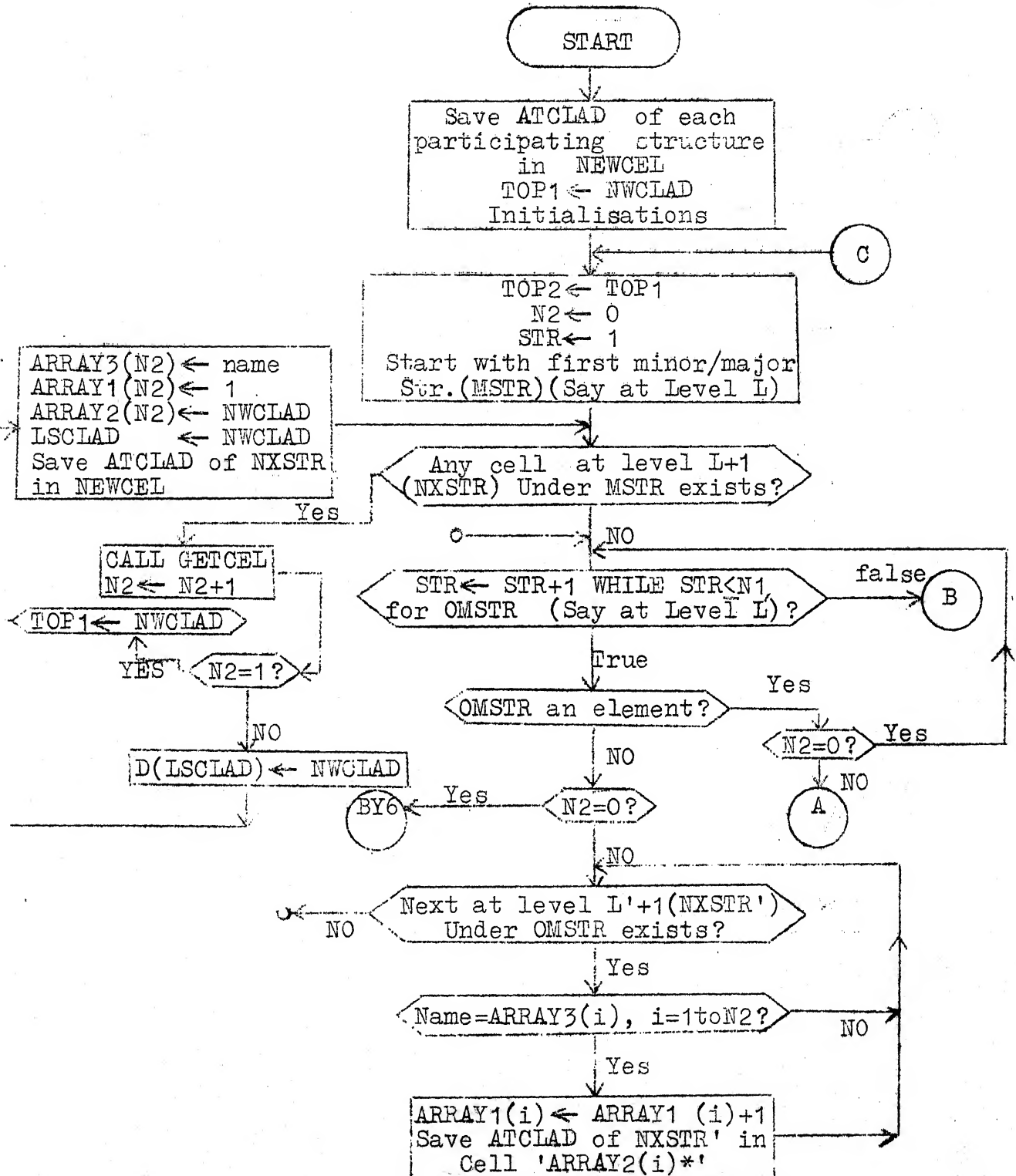
NOMENCLATURE:

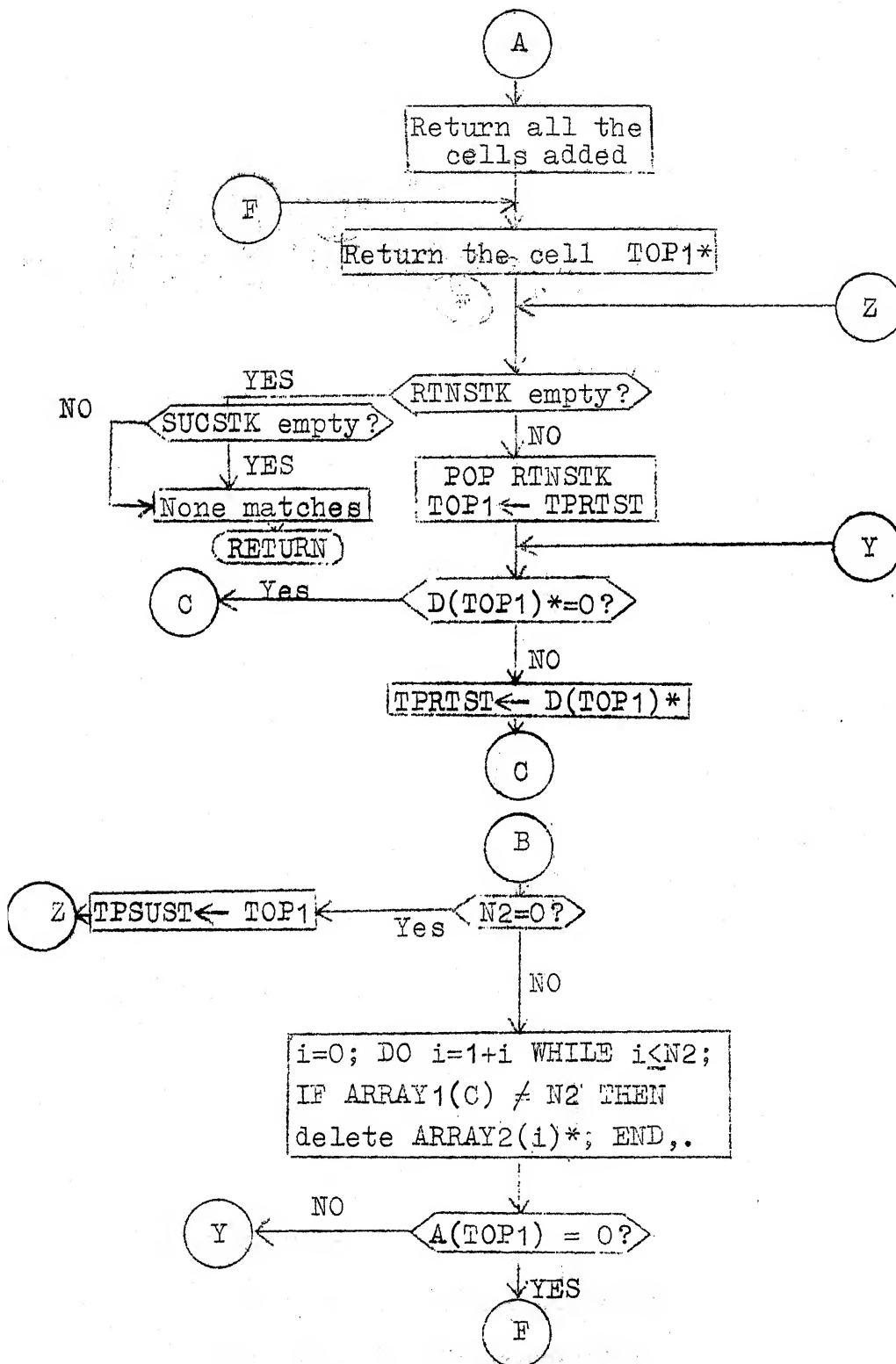
1. Symbols:

ARRAY1:	Stores the 'match' count
ARRAY2:	Stores the added cells address
ARRAY3:	Stores the names of the minor structures at same level under one minor/major structure
ATCLAD:	Attribute cell address
LSCLAD:	Address of six-word cell added last time.
NEWCEL:	New six word cell taken from the free-list
NWCLAD:	Address of NEWCEL
OMSTR:	Other minor/major structures
RTNSTK:	Return address stack
SUCSTK:	Success stack
TPRTST:	Top of the return stack
TPSUST:	Top of the success stak

2. Let word represent name of a computer word then

A(word):	Address portion of 'word'
D(word):	Decrement portion of 'word'
word*:	Represents one level of indirection in 'word'





(BNM-2)

CHAPTER V

PROGRAM CONTROL STATEMENTS

This chapter describes the syntax, semantics and implementation of the following statements:- IF-statement, GOTO-statement, Function reference and CALL - statement, RETURN-statement, BEGIN - statement, END - statement, PROCEDURE - statement, and ENTRY - statement.

1. IF Statement:

1.1 Syntax:

Though the general format and syntax rules have already been discussed in the Chapter on First Pass Processor (FPP), for the sake of continuity these are being enumerated here in brief.

General Format:

IF scalar-expression THEN Unit-1 [ELSE Unit-2]

Syntax Rules:

1. Each "Unit" is a DO-group, a 'begin' block or any statement, other than DECLARE, END, ENTRY, FORMAT, PROCEDURE. The units may have their own labels.

General Rules:

1. When the ELSE clause (ELSE, and its following unit) is not specified, the scalar expression is evaluated and, if necessary,

converted to a bit string. If any bit in the resulting string has the value 1, Unit-1 is executed, and control passes to the statement following the IF statement. If all the bits have the value 0, Unit-1 is not executed, and control passes to the next statement. When the ELSE clause is specified the expression is similarly evaluated. If any bit is 1, Unit-1 is executed, and control passes to the statement following the IF-statement. If all bits have the value 0, Unit-2 is executed, and control passes to the next statement.

The units may contain statements that specify transfer of **control** and so override these normal sequencing rules.

2. IF statements may be nested that is, either Unit-1 or Unit-2, or both, may themselves be IF statements. Each ELSE clause is always associated with the inner most unmatched IF in the same block or DO group, consequently, an ELSE or a THEN with a null statement may be required to specify a desired sequence of control.

1.2 Coding of IF-Statement:

The coding of an IF-statement becomes an easy job because of the work done by the FPP. The FPP supplies the following information to the second pass-

- (i) The scalar expression which is to be evaluated.
- (ii) Success-label: Address of the place where the control should go if at least one bit in the resulting bit string comes out to be 1.

- (iii) Failure-label: Address of the place where the control should be transferred if all the bits in the resulting bit string happen to be zeros.
- (iv) End-label: Address of the place where the control should go after executing Unit-1 in case of success.

The scalar expression is coded by the expression processor, which returns a label to the IF-processor. The coding generated by expression processor would be as follows:

```
[
Code for evaluating the expression,
converting it to a bit string, if
necessary, and bringing the bit string
in the accumulator
]
```

```
TZE Falabel
TRA Salabel
```

where,

"alabel" is the label returned to the IF-processor by the expression processor.

The FPP assigns a unique IF-serial number to each IF-statement. Unit-1 and Unit-2 would be coded like any other part of the program, depending upon the statements involved in the units. The final code corresponding to an IF-statement could be represented, schematically as follows:

```
T.ifno EQU *           generated by 'Success-label'
                        routine

[ Coding for Unit-1 ]

      TRA E.ifno         generated by 'Failure-label'
                        routine
F.ifno EQU *           generated by 'Failure-label' routine

[ Coding for Unit-2 ]
```

E.ifno EQU * generated by 'end-label' routine
 Falabel EQU F.ifno generated by the IF-processor
 Salabel EQU S.ifno

where "ifno" is the IF-serial number.

2. GO TO STATEMENT

2.1 General Format:

$$\left\{ \begin{array}{l} \text{GOTO} \\ \text{GO TO} \end{array} \right\} \left\{ \begin{array}{l} \text{label-constant} \\ \text{scalar-label-variable} \\ \text{reference} \end{array} \right\}$$

GOTO statement causes control to be transferred to a statement which is specified by the destination after GOTO/GO TO. It is used to break the normal sequential flow of the program control.

2.2 General Rules:

1. "Reference" is a reference to a procedure which returns a label constant as its value.
2. A GOTO statement cannot pass control to an inactive block.
 A GOTO statement cannot transfer control from outside a DO-group to a statement inside the DO-group.
3. A GO TO statement that transfers control from one block (S) to a dynamically encompassing block (D) has the effect of terminating block S, as well as all other blocks that are dynamically descendent from block D. The block closing is treated as

normal block closing either by a RETURN statement or END statement.

2.3 Examples and Implementation:

Consider the following hypothetical example:

```
MAIN : PROCEDURE OPTIONS (MAIN);  
      DECLARE LV1(3) LABEL INITIAL (LC1,LC2,LC3),  
              LV2 LABEL, LV3(3) LABEL;  
LV3(1): DO I=1,2,3;  
        GO TO LV1(I); /* GOTO  
          DESTINATION - AN ARRAY ELEMENT */  
LC1 : LV2=LC1;  
      GO TO LC3; /* GOTO DESTINATION -  
        A LABEL CONSTANT */  
LC2:LV3(2): LV2=LC2;  
        LC3: END;  
LV3(3): LV2=LC3;  
        GOTO LABPRC; /* GO TO DESTINATION - RESULT OF  
          REFERENCE */  
LABPRC: PROCEDURE;  
        RETURN (LV3(RNDY1*3.0+1));  
        END LABPRC;  
        GO TO LV2; /* Scalar GOTO */  
        END MAIN;
```

Example GT-1

Some interesting points are brought out by this example.

The destination of the GOTO statements is of the following types-

- i) Label Constant: Such a GOTO is the basic of all, since ultimately all other types of GOTO destinations are transformed to this type.
- ii) Array Element: The value of the array element used in the destination of the GOTO statement. This mode of GOTO acts like a multiway switch and can be thought of as something similar to FORTRAN computed GOTO statement. But

it is much more general than the latter since the correspondence between the computed index and the destination can be changed during run time.

- iii) Reference: This is the most general form of a GOTO statement. Nothing of this type exists in FORTRAN. Consider the example GT-1, in which a procedure is defined (LABPROC) which returns label constants as its value. The label constant returned becomes the destination then.

Though this example can be simulated without using a procedure by changing the statement

GOTO LABPROC; by GOTO (LV3(RNDY(1)*3.0+1));

yet it serves to illustrate the focal point. The procedure LABPROC calculates the subscript to be used with the label array element. The calculation of subscript is effected by calling a built-in, random number generating routine. RNDY1 returns a uniformly distributed random number in the range $0 \leq x < 1.0$.

In fact any procedure can be invoked as long as it returns a label constant as its value.

2.3.2 Implementation:

For implementing, GOTO statement has been classified under following classes:

1. Ordinary GOTO (destination a label-constant). It can be further subclassified as, when

- a) No block closing and no do closing is involved.
- b) Only block closing is involved.
- c) Only do closing is involved, and
- d) Both block closing and do closing are involved.

2. Scalar GOTO (destination either a normal label variable or a formal label variable).

3. Expression GOTO (destination either an array element, a function reference or a label expression).

Label expression is an expression whose value is a label constant. Example of label expression

```

    DECLARE BIT BIT;
    L1:S-1;
    L2:S-2;
    .
    .
    .
    GOTO (BIT*L1 +(NOT BIT)* L2);
    .
    .
    .

```

(This type of GOTO is prohibited in this version.)

(a) Ordinary GOTO:

Consider the following example,

```

MAIN  : PROCEDURE OPTIONS (MAIN);
        DECLARE C(10);
D1    : DO I=1,5 TO 10 BY 1; /* FIRST DO-GROUP */
        GO TO L1; /* NO BLOCK/DO CLOSING */
L22:L2: A=C(I)
D2:    : DO J=1 TO 10 BY 2; /* SECOND DO-GROUP */
L3:L9 : A=C(J); GOTO L2; /* ONLY DO CLOSING */
B2    : BEGIN;
        GOTO L3; /* CLOSING BLOCK CLOSING */
        .
        .
        .
        GOTO L22; /* BOTH BLOCK AND DO CLOSINGS */
        END D2;
L1    : A=1.0
        END MAIN;

```

Example GT-2

Definition: Source: Place where the GOTO statement occurs.

Source has attributes of block number and DO number.

(i) NO Block/do closings:

Consider the statement 'GOTO L1;'. It transfers the control to a statement which is in the same block and in the same do-group as the source.

Coding of such a GOTO is the simplest -

TRA label

where ,

"label" is the destination address.

(ii) Only Block Closing:

Consider the statement "GOTO L3;".

Both the source and the destination are internal to the same do and hence have the same do number, but their block numbers are different. Source is internal to B2, whereas the destination is internal to MAIN. In this case block B2 would have to be closed before transfer can take place.

Code generated for such a case is,

```
TSX  GTBLOK,4
PZE  label
PZE  sblno,,dblno
```

where,

"sblno" is the block number of the source, and

"dblno" is the block number of the destination.

The GTBLOK is a part of "run time stack management routines" (Appendix-0). It goes on closing the blocks till it reaches the destination block. During this process, if more than one generation of a RECURSIVELY called procedure exist and control is to goto a dynamically encompassing block, all the generations are terminated.

While terminating the blocks, till the destination block is reached, if run-time stack gets emptied, it indicates an error; the destination is inside a block which is not open when the GOTO statement is being executed.

(iii) Only DO Closing:

Consider the statement 'GOTO L2;'

Though both the source and the destination are internal to the same block (MAIN), yet they belong to different DO-groups. Source is internal to the second DO whereas the destination is internal to the first DO.

A DO Status Table (DST) is maintained at run time which keeps the Do-predecessor relationship and Do-Status (Open/Close) information for every DO defined in the program, except the DO's used in Input/Output statement. DO status bit is ON, if that particular DO-group is open. The do number of the inner most DO group open is kept in a location called DOSRNO. The DST is made use of for determining the illegal transfer to inside a closed DO from outside the DO.

The code generated is as followss

```
TSX  GTDOBL,4
PZE  label
PZE  s-dono,,d-dono
```

where,

" s-dono " is the do number of the SOURCE, and

" d-dono " is the do number of the DESTINATION.

The routine GTDOBL follows the do predecessor relationship till it reaches the destination DO group. If the status bit of the destination DO is OFF, it indicates illegal transfer from outside the DO to inside the DO. The do status bits are updated every time a DO-group is opened or closed.

(iv) Both Block Closing and DO Closing:

Consider the statement 'GOTO L22;'

The source is internal to the block B2 and the DO group D2 whereas the destination is internal to the block MAIN and the DO group D1.

The code generated for this type of GOTO is

```
TSX  GTDOBL,4
PZE  label
MZE  s-dono,,d-dono
PZE  s-blno,,d-blno
```

It would be noted that GTDOBL was called for the case (iii) also when no block closing was involved. To distinguish between the two calls, sign of the third word in the calling sequence is made use of.

This call is executed in two parts. First GTBLOK is called to close the blocks and then GTDOBL is called (via first calling sequence) to close the DO-groups.

b) Scalar GOTO:

Consider the statement 'GOTO LV2;'

where LV2 is a scalar variable (normal). The code generated for this case is

```
TSX  GTSCLR,4
pfx  fstaddr
```

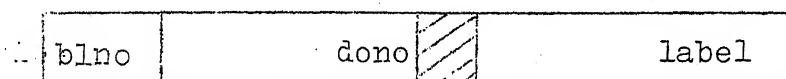
where,

"fstaddr" is the first order storage address of the variable.

" pfx " = PZE for normal scalar

MZE for formal scalar

The routine GTSCLR determines whether any block closing or DO closing is involved or not. This is done by consulting DOSRNO, PSRNO. and the value of the scalar, which contains block number, DO number and the address where to go in the following format -



The global location PSRNO. contains block number of the latest block open and DOSRNO contains the DO number of the inner most DO open. Depending upon the four possibilities it branches to the four routines described in section (a) with ordinary GOTO.

(c) Expression GOTO:

Expression processor is called to process the expression and to generate the code for evaluating that expression at run time. The value of the expression is brought into the accumulator. Then the code generated would be as follows:

<p>Coding for getting the value of expression in Accumulator</p>
--

TRA GTARIT

The routine GTARIT acts as follows:

```

        STO  TEMP
        TSX  GTSCLR,4
        PZE  TEMP
TEMP    BSS   1

```

i.e. it stores the value of the expression in a location called TEMP. Then expression GOTO becomes a scalar GOTO.

3. Function Reference and Call Statement:

3.1 SYNTAX:

General format of function reference:

```
{entry name|generic name} [(argument [,argument]...)];
```

General format of CALL statement:

```
CALL {entry name|generic name} [(argument
                                [, argument]...)];
```

where "argument" is a scalar expression, file name or label constant.

A procedure block can be invoked either by a CALL statement or by appearance of its name. The latter is called 'function reference' whereas the first one is called 'subroutine reference'. The essential difference between the two is that while 'function reference' expects a value to be returned as a result of procedure invocation, CALL does not expect any result.

In case a generic name is used, the compiler selects one of the many members belonging to this generic family, depending on the attributes of arguments used in the argument list.

3.1.1 CODING OF Function Reference and CALL Statement:

The list of arguments following the procedure name is processed by the expression processor which generates a coded list (argument-list) and outputs it under the location counter 'JUNK' in order to bypass the normal sequential flow of the program. If any constant is used as an argument, it is assigned to a temporary and the temporary address is supplied to the argument list. Argument list has one word for each of the arguments which stores the first order address of the variable. In case of an expression being the argument, the expression is evaluated, result assigned to a temporary and address of the temporary filled in the argument list.

If a formal scalar is being passed as an argument, special care is to be taken while preparing the argument list. Since formal parameter involves one level of indirection for some scalars (Appendix D), the address contained in the locations assigned to the formal parameter should be filled in the argument list, and not the address of the formal parameter storage space. This becomes necessary because the prologue list processing routine when it consults the argument list, has no way of finding out whether a formal parameter is being passed on or a normal variable is being passed on as the argument.

The code generated for invoking a procedure is as follows:

For Function Reference:

```
TSX  R.OPPR,4
TXI  *+3,,arglst
PZE  PHWadr
PZE  temp,,mntyp
```

For CALL Statement:

```
TSX  R.OPPR,4
TXI  *+3,,arglst
PZE  PHWadr
PZE  0
```

where,

" arglst " is the address of the argument list,
Zero, if there are no arguments,

" PHWadr " is the address of the procedure header word,

" temp " is the result-receiving temporary address,

" mntyp " is minor type of the result expected, (Appendix-D)

R.OPPR is a routine which opens a procedure when invoked either by function reference or a CALL statement, makes the link cell and does the prologue initialisation. For details reference should be made to Appendix-0.

Fourth word in the calling sequence is used to distinguish between two types of calls. If fourth word is zero, it indicates a call from CALL-statement otherwise from function reference.

3.2 RETURN-Statement:

3.2.1 General Format:

```
RETURN [(scalar expression)];
```

The RETURN statement can have two options:

Option-1:

RETURN;

Option-2:

RETURN (scalar expression);

3.2.2 General Rules:

1. Only the RETURN statement of option-1 can be used to terminate procedures not invoked as function procedures; control is returned to the point logically following the invocation.
2. The RETURN statement in option 2 can be used to terminate a procedure invoked by a function reference as well as by a CALL statement. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified in RETURN, converted to conform to the attributes declared for the invoked entry point. These attributes may be specified explicitly at the entry point, they are otherwise implied by the initial letter of the entry name by which the procedure is invoked. If the type of the result being returned does not match with the type that was expected at the time of making the call, necessary conversion is made, if possible.

3. If control reaches an END statement corresponding to the end of a procedure, this END is treated as a RETURN statement (of the option-1) for the procedure.

3.2.3 CODING of RETURN-Statement:

For Option-1:

```
TSX CLPRWR,4
```

For Option-2:

```
Code for evaluating the expression,
if any and for assigning the result
in a temporary
```

```
TSX CLPRRT,4
PZE temp,,mntyp
```

Code for evaluating the scalar expression, if any is generated by the expression processor. The result of the expression is assigned to a temporary. In case there is no expression but only a constant is being returned, it is outputted and its address is plugged in place of 'temp'. Similarly, if contents of a scalar variable are being returned as result, address of the first/storage area allocated to that variable is plugged in place of 'temp'. For a formal scalar involving one level of indirection, the contents of that word are plugged in place of 'temp'.

Both the routines CLPRRT and CLPRWA are part of the 'Run Time Stack Management Routines'(RTSMR) and are explained in Appendix-0. They do all the book keeping operations and serve as epilogue.

3.3 BEGIN-Statement:

3.3.1 General Format:

BEGIN;

3.3.2 General Rule:

A BEGIN Statement is used in conjunction with an END Statement to delimit a begin block.

3.3.3 Coding for BEGIN Statement:

```
AXT blkopn,1  
TSX R.OPBG,4
```

where 'blkopn' is the block serial number of the block being invoked. R.OPBG routine, again, is a part of RTSMR. The job assigned to this routine is mainly for making a linkage cell. Unlike a procedure invocation, which occurs with break in sequence, begin block is invoked, in the normal sequence, when control comes to the BEGIN-statement.

3.4 END-Statement:

3.4.1 General Format:

END [label];

The END statement terminates blocks and DO-groups.

3.4.2 General Rules:

1. If a label follows END, the END statement terminates the unclosed block or DO-group that is headed by the nearest preceding heading statement having that label; it also terminates all

unclosed blocks and DO-groups that are lexicographically within that block or group.

2. If a label does not follow END, the END statement terminates that group or block headed by the nearest preceding DO; BEGIN or PROCEDURE statement.

3. If control reaches an END statement, terminating a procedure, it is treated as a RETURN statement.

The detection of 'multiple closing' and classification of END as to whether closing a DO-group, BEGIN-block or PROCEDURE-block is done in first pass itself. In case of multiple closure first pass supplies as many ENDS as necessary, each with sufficient classifying information.

3.4.3 Coding of END Statement:

For END closing a BEGIN/PROCEDURE block:

TSX CLSEND,4

For END Closing a DO-group:

TSX CLOSDO,4

Both the routines are parts of RTSMR. In case this END statement is being used as a RETURN statement, CLSEND merges with the CLPRWR routine. Otherwise it removes the link cell from the run time stack and updates the block-open number.

It then returns control to the calling point to execute the next sequential instruction.

The function of CLOSDO routine is to put off the DO-on-flag corresponding to the DO number of the DO-group being closed. It also updates the DOSRNO, which contains the serial number of the DO which is open. In case there is no such DO, DOSRNO is made zero.

4. Procedure And Entry Statement:

4.1 SYNTAX:

ENTRY statement:

```
{entry name:} ....
ENTRY [(parameter[,parameter]...)]
[OPTIONS (option list)]
[RETURNS (data-attribute list)];
```

PROCEDURE statement:

```
{entry name:} ...
PROCEDURE [(parameter [,parameter]...)]
[OPTIONS (option-list)]
[RECURSIVE]
[RETURNS (data-attribute list)]
[ORDER|REORDER];
```

4.1.1 SYNTAX Rules:

1. The OPTIONS, RECURSIVE, RETURNS and ORDER|REORDER options may appear in any order.
2. The syntax of OPTIONS-list is not restricted in PL 7044. In fact it is completely ignored by this compiler, as also all

other attributes except RETURNS. They are accepted for the sake of compatibility.

3. Any procedure can be invoked recursively irrespective of the fact whether RECURSIVE attribute is defined or not.

4.1.2 General Rules:

1. The 'parameters' are names that specify the parameters of the entry point. When the procedure is invoked a relationship is established between the arguments of the invocation and the parameters of the invoked entry point.

2. Only arithmetic attributes can be used inside RETURNS option. The data attributes specifying string-length attributes are ignored.

3. An ENTRY statement cannot be internal to a BEGIN block. PROCEDURE and ENTRY statements cannot be internal to a DO-group.

4. If an ENTRY/PROCEDURE statement has one or more than one label attached to it but does not specify RETURNS specifications, defaults are applied separately to each name, depending on the initial letter (or/letter) of the identifier, and to each returned value as if it had an identifier beginning with same letter (or/letter) as the corresponding entry name.

Consider the statements

B:J:PROCEDURE RETURNS (FIXED (5,0));

⋮
A:I:ENTRY;

⋮

Function reference to either of B or J would return an integer value, whereas function reference to A would return a floating point constant & function reference to I would return an integer value, though both of them refer to the same entry point.

5. An extra restriction has been put on the use of DECLARE statements. All DECLARE statements should be put immediately after the PROCEDURE or ENTRY statement. In fact only specifications, and not declarations should be put after ENTRY statement.

This restriction solves many complicated logical problems. Further comments on this would be made at the end of this chapter.

4.2 Prologue and Parameter List:

On entering a block certain initial actions are performed i.e. allocation of storage for automatic variables etc. Each block consists of three parts -

- a) Prologue
- b) Text, and
- c) Epilogue.

Prologue does all the initialisations before entering the text portion.

Text portion represents the actual code of the block.

Epilogue does all the work of resetting, once a block is terminated.

At the beginning of the prologue, the following items are available for computation:

1. The established generation of automatic variables declared outside the block and known within it.
2. Static variables known within the block.
3. Arguments passed to the block.

The prologue makes available for computation all the variables known within the block as follows. In making AUTOMATIC variables declared in the block available the prologue may need to evaluate expressions concerned with automatic data. Such expressions may occur specifying lengths, bounds, size of arrays and iteration factor. However, a circular definition should be avoided. It constitutes an error which can not be detected by our compiler, e.g. following declaration is illegal,

```
DECLARE A(M(I)), M(A(J));
```

The prologue in a conventional compiler viz. IBFTC of IBM-7044, is put along with the procedure. The prologue of PL-7044 is divided into two parts. The job of making arguments available to the procedure being invoked, is taken away from the procedure, and is delegated to the calling block. The job of assigning area for automatic variables is, however, still with the procedure.

For every entry name, one word is outputted, in the code, which is called the Procedure Header Word (PHW). PHW contains the following information -

- i) Prologue list address (prglst)
- ii) Address of the entry point where transfer should go (ent add).
- iii) Block number of the block to which the text of this procedure belongs.

The format of PHW is as follows -

X	prglst	Y	ent add
---	--------	---	---------

where, YX is the block number (in OCTAL characters) to which the text belongs. e.g. for a block with number 13, Y = 1 and X = 5.

Prglst is zero if there is no argument for this procedure reference.

While invoking a procedure, PHW is consulted by the run time block invoking routine to determine the things listed above.


PHW is transmitted, if an entry name is used as an argument.

4.2.1 Prologue List:

Prologue list is generated at the compile time, while processing the specifications for formal parameters. Prologue

list stores the first order storage address, parameter number and the type of that parameter. It is then used for making the argument list known to the procedure.

The prologue list has one word for each of the parameter that is not a major structure. For each base element of the major structure, two words are used in the prologue list. The format of the prologue list word is as follows -

	fst add	PARMNO	Major type	5
---	---------	--------	---------------	---

where "fst add" is the address of the first order storage (in case of multiword, it represents the address of the first word) assigned to the parameter.


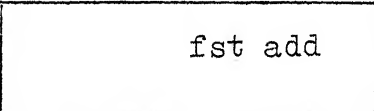
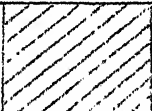
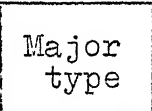


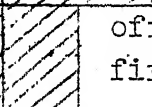
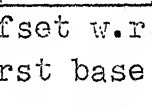
PARMNO is the serial number of the corresponding parameter.

Major type is one of the following:

<u>Major type</u>	<u>Description</u>
1	integer/floating point/complex array
2	not applicable (Picture integer/floating point array)
3	character string array
4	Bit string array
5	Label array
6	not applicable (non numeric picture array)

<u>Major type</u>	<u>Description</u>
7	integer/floating point scalar
8	complex scalar
9	not applicable (non numeric picture scalar)
10	not applicable (Picture integer/floating point scalar)
11	label scalar
12*	file constant
13	bit string scalar
14	bit string (varying length) scalar
15	character string scalar
16	character string (varying length) scalar
17*	major structure
18*	entry constant

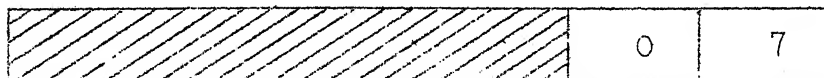
The format of prologue list words for a base element of a major structure is as follows:

			fst add		Major type	7
				offset w.r.t. first base element		

where,

Major type is one of those given above except the ones marked with an asterisk, since they can not occur as base elements of a structure.

The base elements of a structure cause the generation of two consecutive words. The base element definition for a structure is ended by the following word



The entire prologue list for one procedure or entry statement is terminated by the following word



All the prologue lists are generated under one location counter called 'PROLOG' to keep them out of the normal sequence of execution. The prologue list generated for the following example would serve to illustrate the main points.

```

E: PROCEDURE (A,B,C);
  DECLARE A COMPLEX, 1 C, 2 D, 3 E REAL, 3 F (10),
          2 G CHARACTER;
  <statement other than a declare statement>
  .
  .
  .

```

Prologue List Generated:

```

      USE  PROLOG
L00001 EQU  *
      VFD  3/0,15/BS.02+000000,6/1,6/8,6/5
      VFD  3/0,15/BS.02+000002,6/3,6/17,6/5
      VFD  3/0,15/BS.02+000003,6/0,6/7,6/7
      OCT  000000000000
      VFD  3/0,15/BS.02+000004,6/0,6/1,6/7
      OCT  000000000001
      VFD  3/0,15/BS.02+000005,6/0,6/15,6/7
      OCT  000000000002
      PZE  7
      VFD  3/0,15/BS.02+000006,6/2,6/7,6/5
      PZE  0

```

It is assumed that block number of the block is 2.

'BS.02+ offset' represents the first order storage address.

It should be noted that prologue list is not in the same order in which parameters appear in the PROCEDURE/ENTRY statement, because the order of their appearance in prologue list depends upon the sequence of declaration. In this particular case first word is for A, followed by 8 words for the structure C and one word for B, which is given default attributes since no declaration is given for it. The last word indicates the end of the prologue list. L00001 represents the address of the prologue list for this particular case.

4.3 Coding of PROCEDURE and ENTRY Statements:

4.3.1 Coding of PROCEDURE Statement:

```

          TSL  DCL.bn
          TRA  EDC.bn
DCL.bn   PZE  **

```

Code for array allocation expression evaluation for string length, initialisation etc. Code for doing label initialisation of prefix type

```

          TRA* DCL.bn
EDC.bn   EQU  *

```

where, bn is block number.

This scheme of putting code for storage allocation and initialisation out of normal sequence becomes necessary to take care of invocation of the same procedure at secondary entry points, because in that case also all this work would have to be done before starting the actual text since it is part of the prologue.

Reason for not allowing ordinary declarations to appear after an ENTRY statement is precisely this. If it is allowed, another small chunk of code would have to be generated out of the normal sequence. Since there is no restriction on number of secondary entry points that can be defined, there might be many such isolated pockets of code lying in the program.

It is not that it cannot be taken care of, but at the expense of both storage space and execution time. A chained structure, like label initialisation of prefix type, can be maintained which can be followed in exactly the same manner as the other list. Since label initialisation of prefix type would not be used that frequently, the resulting inefficiency can be tolerated but not in the present case because it would be used in almost every program.

4.3.2 Coding of ENTRY Statement:

TSL DCL.bn

The requirement that all the specifications be put immediately after the ENTRY statement or PROCEDURE statement is necessary to end a particular prologue list before the other is started. This becomes necessary for another reason also. Since all parameters need not be specified explicitly in a DECLARE statement, prologue list entries would have to be generated for such cases, corresponding to default attributes.

Like the case of declarations, a chained prologue list is alternative to the restriction, but once again, increased storage requirements and increased execution time because of chaining weigh in favour of the arrangement followed

APPENDIX A

A-1: Character Sets:

For writing the source program in PL-7044, 48-character set is to be used. The 48 - Character set is a subset of the 60 - character set. The 48- character set comprises of 26 alphabets, 10 digits, and 12 special characters.

The special characters are,

<u>Character</u>	<u>Representation</u>
Blank	
Equal sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis)
Comma	,
Point or period	.
Single quotation mark	'
Dollar symbol	\$

To represent the remaining characters of the 60 - character set, which are used in PL/I language, composite symbols are used. The list of composite symbols and the equivalent 60 - character set representation is given in the following table.

Composite Symbols C	60-character set	
<u>Composite Symbols</u>	<u>Equivalent</u>	<u>Name</u>
..	:	Colon
,.	;	Semicolon
OR		'Or' Symbol
AND	&	'AND' Symbol
GT	>	'GREATER THAN' Symbol
LT	<	'LESS THAN' Symbol
NOT]	'NOT' Symbol
LE	<=	'LESS THAN EQUAL TO' Symbol
CAT		'CONCATENATION' Symbol
**	**	'EXPONENTIATION' Symbol
NL] <	'NOT LESS THAN' Symbol
NG] >	'NOT GREATER THAN' Symbol
NE] =	'NOT EQUAL TO' Symbol
//	%	'PERCENT' Symbol
GE	>=	'GREATER THAN EQUAL TO' Symbol
/*	/*	'START OF COMMENT' Symbol
*/	*/	'END OF COMMENT' Symbol.

With 48- character set, following rules are to be observed:

1. The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.
2. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk, or immediately followed by a blank if the following character is an asterisk.

3. The sequence " Comma Period" represents a semi colon except when it occurs in a comment or character string.

A2: LEXICAL REPRESENTATION OF VARIOUS LEXUNITS:

Operators



where,

"cp" is compare priority, and

"descp" is the description number which uniquely identifies an operator and is used in table look up etc.

"descp" and " cp" for various operators are as follows:

<u>descp</u>	<u>cp</u>	<u>Operator</u>
1		Unary minus (assigned by the expression processor)
2	11	NOT
3	11	**
4	10	*
5	10	/
6	9	+
7	9	-
8	7	CAT
9		Relational '=' (EQ) (Assigned by the expression processor)
10	6	NE
11	6	GT
12	6	GE
13	6	NG
14	6	LT
15	6	LE
16	6	NL

Contd.

17	5	AND
18	4	OR
19	4	,
20	3)
21	11	(
22	11	=
23	2	..
24	4	..
25	-	.

Operands:

Unlike operators, operands contain both the type information and their description. Type is available in a word called TYPE1, whereas the description is available in other associated words. The actual operand is collected in a six word buffer called LEXBUF.

type	TYPE1
------	-------

where "type" is
type

Operand

1	Pseudo reserved word or Key word
2	Identifier or non Key word
3	Integer Constant
4	Floating point constant
5	Bit string constant
6	Character string constant

Pseudo Reserved Words:

These represent identifiers having special meaning. The actual name string is available in words starting with LEXBUF and word count and character count is available in WCOUNT and CCOUNT respectively. The code number to identify a particular pseudo reserved word is available in CODE.

Identifiers:

These are symbols which have no special meaning. Their description is exactly like pseudo reserved words except that they have no code numbers.

Integer/Floating point constant:

Actual constant, converted from BCD to Binary, is available in LEXBUF.

Actual bit string, left justified, is available in LEXBUF and bit count is available in CCOUNT.

Character String Constant:

Actual character string is available in words starting with LEXBUF, and word count and character count is available in WCOUNT and CCOUNT respectively.

APPENDIX B

Constants in PL-7044:

Constants that can be used in PL 7044 are of the following five types:

1. Arithmetic constants,
2. String constants,
3. Statement label constant,
4. Entry constants, and
5. File constants.

Arithmetic Constants:

Arithmetic constants can be one of the three types.

a) Integer Constants:

Maximum length of integer constants allowed in 27 bits i.e. an integer greater than $2^{27}-1$ (163708927) can be used.
Examples: 121, 0, 7777 etc.

b) Floating Point Constants:

Floating point constant of PL 7044 is exactly like the floating point constant of IBM 7044 - FORTRAN i.e. the range is 10^{-38} to 10^{+38} with 8 decimal digit precision.

Examples: 1.23, 1.0 E-10, .0003, 5E2 etc.

c) Complex Constants:

Complex constants can be of two types

- i) purely imaginary constant e.g. $\pm 2I$ and
- ii) Complex constants having real and imaginary part both, e.g. $1 \pm 2I$.

Complex constant is stored as real constant and imaginary constant. Both these constants are floating point constants. The internal representation of the two constants shown here would be $0 \pm 2.0I$ and $1.0 \pm 2.0I$ respectively.

String Constants:

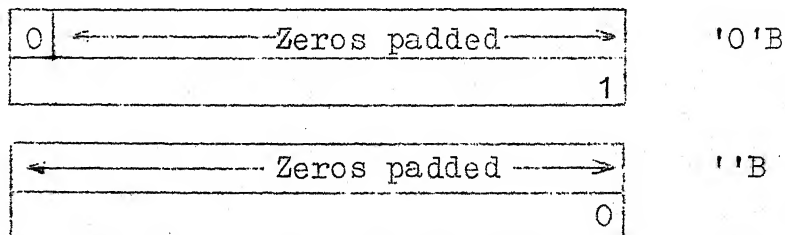
String constants are of two types:

a) Bit string constants:

Each digit can be either 1 or 0 only. Its representation is as follows -

'101'BB, '0'B , ''B

The bit strings '0'B and ''B are different, first being a bit string of length 1 whereas the latter is a null bit string. Maximum length of a bit string can be 36 (word size of IBM 7044). Bit string is stored left justified with remaining bit positions filled with zeros. Each constant at run time would be stored in two consecutive words, first word containing the actual bit string and the second word contains the bit string length. The representation of '0'B and ''B would be as follows



b) Character String Constants:

A character string can be constructed using any of the 48 characters of the 48-character set. The representation of a character string is as follows:

'GARIBI HATAO'

'C.I.=PR*(1.+RT/100.):**NY-PR'

' '

''

The last two character strings are not same. First of the two is a character string of length 1 containing one 'blank' character. The last one is a null character string i.e. string of length zero. The length of character string represents the number of characters in the character string. Maximum length of a character string constant can only be 30.

If quote happens to be one of the characters of the string then it should be repeated i.e. two quotes for each quote that is a part of the character string e.g.

TEXT

IT WON'T HAPPEN

" DON'T GO", SHE SAID

''

REPRESENTATION

'IT WON"T HAPPEN'

""DON"T GO,"" SHE SAID'

''''

Each character string constant is stored in consecutive words with remaining right most character positions of the last word, if any, padded with blanks (Ø60 in IBM 7044). The first word in the storage for a character string constant, stores its length.

					4
S	T	O	P		
					11
'	G	O		T	O
	H	E	L	L	'

'STOP'

'GO TO HELL'

Repetition Factor:

The representation of a long string containing a repetitive substring can be made quite concise by using 'repetition factor' e.g.

Expanded formShort form

'1111111'B

(7)'1'B

'ABCABCABC'

(3)'ABC'

'10101010'B

(4)'10'B

The length restriction of 30 characters holds for the expanded string constant. Thus expanded bit string constant should not use more than 36 bits while the expanded character string constant should not be more than 30 characters long.

Statement Label Constants:

Statement label constants are the symbols prefixed to any statement except the PROCEDURE and ENTRY statements. e.g. consider the following statements -

```

COMPUTE:  A = B+C;
BRANCH:  GO TO HELL;
LAB1:LAB2:LAB3:DO;

```

In the above example COMPUTE, BRANCH, LAB1, LAB2, LAB3 are statement label constants. The constructed label constant format is as follows

bl. no.	Do-number		Label Address
------------	-----------	--	---------------

where,

"blno." is the serial number of the block to which the statement appended to the label constants, is internal.

"Do-number" is the serial number of the immediately encompassing do-group. In case no such do-group exists, it is zero.

"Label Address" is the starting address of the code generated for that statement.

This constructed label constant is used in label assignment.

Entry Constants:

Entry constants are symbols prefixed to the PROCEDURE and ENTRY statement. In the following example FACTORIAL, MAIN, INTEGER, COMPLEX and FLOATINGPOINT are examples of entry constants.

```
FACTORIAL: PROCEDURE;
MAIN: PROCEDURE OPTIONS (MAIN);
COMPLEX: FLOATINGPOINT:INTEGER:
          PROCEDURE(X) RETURNS (REAL(10,0));
```

Since entry variables are not allowed in PL 7044, constructed entry constant is not defined. However since entry constants can be passed as arguments, a procedure-header-word is defined with every entry constant, whose format is as shown below,

x	prglst address	y	text address
---	----------------	---	--------------

where,

"yx" is the serial number of the block to which the code belongs.

" prglst address" is the address of the prologue list generated, if any, for the entry point, otherwise it is zero.

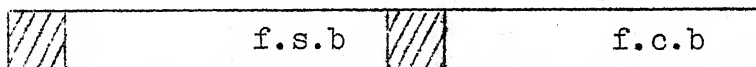
" text address" is the address where the control should be transferred at the time of invocation.

File Constants:

File constants are names given to files which are then referred to by these names, e.g.

```
DECLARE INPUT FILE;
OPEN INPUT;
```

INPUT is a file constant here, constructed file constant representation while transmitting as an argument is,



where,

f.c.b. is file control block address and

f.s.b. is file status block address.

APPENDIX C

Names in PL 7044:

Every variable defined in PL/I is given an alphameric name so that a reference can be made to it by this name. Names can be of two types:

- i) Non qualified or Simple names
- ii) Qualified names

Non Qualified Names (NQ-name):

Following rules should be observed while constructing a NQ-name:

- 1i) first character (or the only character) of a name should be one of the 26 alphabets.
- ii) Remaining characters can be any of the 26 alphabets and 10 digits.
- iii) More than 30 characters should not be used in forming a name.
- (iv) There should not be any intervening blanks since blank serves to delimit a name.

Valid names:

BONDOO7, APOLLO, I101, J

Invalid names:

1A, BOND.1, RAM LAL

Qualified names (Q-name):

A Qualified name can be defined as

NQ-name { .NQ-name } ...

A qualified name is used to refer to minor structures or base elements of a structure.

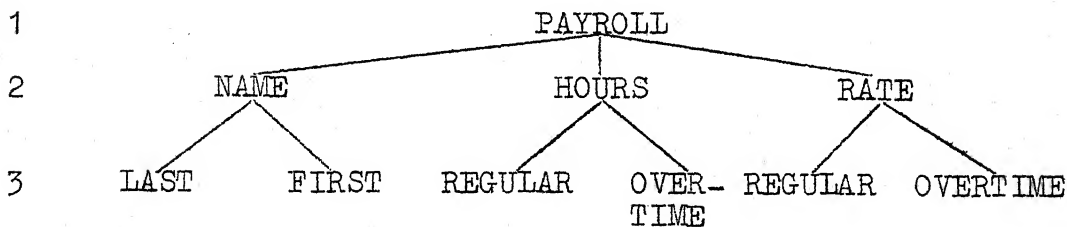
Consider the following structure:

```

DECLARE 1 PAYROLL,
        2 NAME,
          3 LAST,
          3 FIRST,
        2 HOURS,
          3 REGULAR,
          3 OVERTIME,
        2 RATE,
          3 REGULAR,
          3 OVERTIME;

```

The tree corresponding to this structure would be



Qualification in the Q-name is in the order of levels; that is the name at the highest level must appear first, with the name at the deepest level appearing last.

Since any of the names in a structure, except the major structure name itself, need not be unique, Q-name makes it unique. Thus in the above example only REGULAR is ambiguous but PAYROLL.HOURS.REGULAR and PAYROLL.RATE.REGULAR are not. However, names of all the nodes from root to the node in question, need not be used in qualification, if it does not give rise to ambiguity. Though LAST and FIRST uniquely represent the concerned nodes, PAYROLL.NAME.LAST and PAYROLL.NAME.FIRST represent the complete qualifications.

APPENDIX D

D-1: Variable Types in PL-7044:

Following types of variables are provided in PL 7044,

1. Arithmetic variables
2. String variables
 - a) Bit string variables
 - b) Character string variables
3. Label variables

Notation: 'varadr' represents the address of the first order storage assigned to the variable, and is available in the symbol table.

Arithmetic Variables:

Arithmetic variables are names of arithmetic data items. These names have been given the characteristics (i.e. attributes) of Base, Scale, Mode and Precision. PL 7044 has only BINARY base though it accepts DECIMAL, if specified yet BINARY is implied. This is because IBM 7044 has no DECIMAL arithmetic hardware. Similarly PL 7044 accepts both FIXED point and FLOATing point scales, with precision specified, for the sake of compatability but internally it works with only two representations - Floating point (8 bit characteristic and 2 bit mantissa in normalised representation) or Integer (27 bit).

A specification of type FIXED (m,0) or FIXED (m) implies integer representation in PL 7044 whereas FIXED (m,n), $n \neq 0$ and FLOAT (m,n) implies floating point representation.

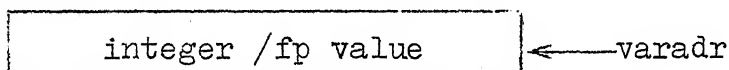
Output from PL 7044 compiler is only in these modes though input is accepted in the general format, again for compatibility sake.

Thus arithmetic variables can be of three types -

1. Integer,
2. Floating point, and
3. Complex.

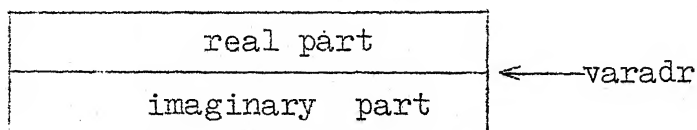
Integer and Floating Point Variables:

Each integer or floating point word requires one storage word. In case of formal parameters of these types, this word points to the parent integer or floating point variable passed as the argument i.e. one level of indirection is involved.



Complex Variables:

Each complex variable is assigned two consecutive words in the storage. In case of a complex formal parameter, both these words point to the corresponding words of the argument passed. Thus both real and imaginary parts involve one level of indirection.



String Variables:

String variables are names of string data items. These names have been given string attributes. The general format of defining string variables is,

name name { BIT
CHARACTER } [(length)] [VARYING]

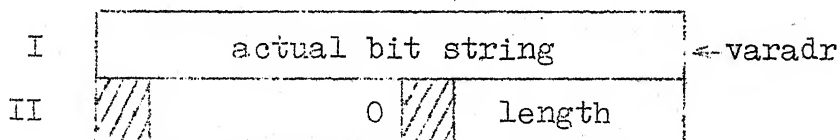
The length attribute specifies the length of a fixed length string or the maximum length of a variable length string. When the length attribute is omitted a length of 1 is assumed.

While there is no restriction on the length of a character string variable, a bit string variable length can not be more than 36 (IBM-7044 word size).

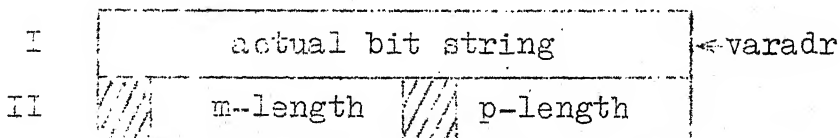
The VARYING attribute specifies that the variable is to represent varying length strings. The current length at any time is the length of the current value.

BIT STRING:

Like complex variable, a bit string variable also requires two consecutive storage words. The format is



Fixed Length Bit String.



Varying Length Bit String.

where,

"m-length" is maximum length, and

"p-length" is present length.

The reason for making the sign of the header word negative would be explained shortly.

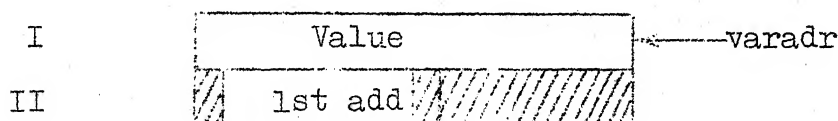
For a formal parameter of character string variable type, no special treatment is given, since the header word of the argument is copied in the parameter storage word and then it becomes exactly like that of a normal variable.

Label Variable:

A label variable is a variable that has as values, statement label constants (slc). The attribute specification may include the values that the name can have during execution of the program. General format is,

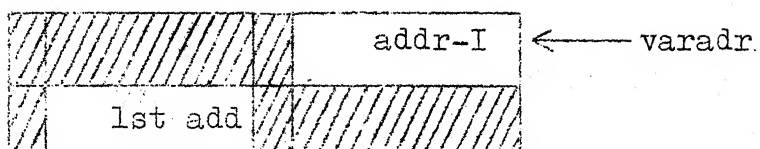
LABEL [(slc [, slc]...)]

Label variable is given two words from the first order storage area. The representation is as follows,



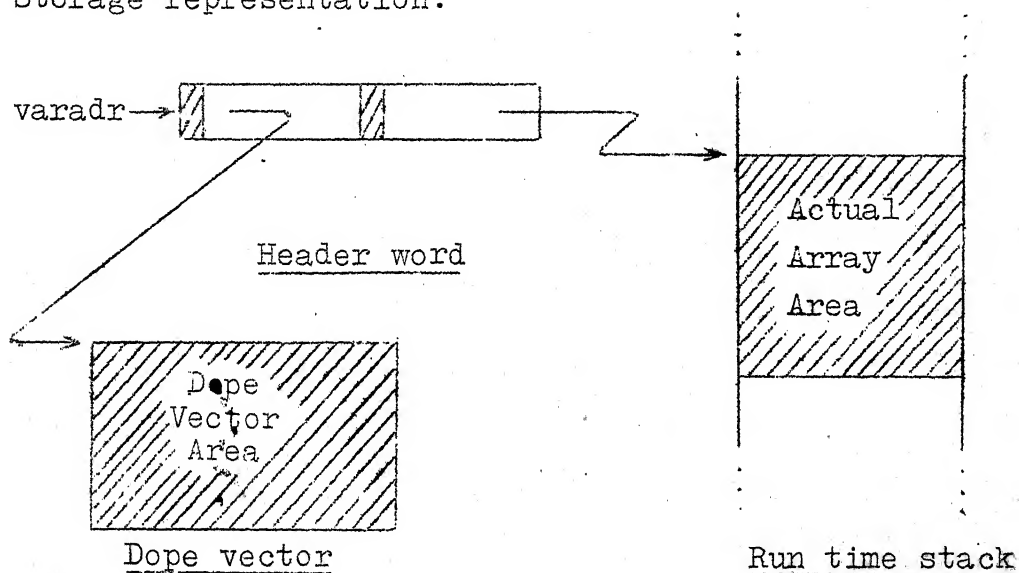
where "1st add" is the address of the label-list, if one specified otherwise it is zero. Label-list is an array of constructed-label-constants generated from the list specified after the keyword LABEL.

For a formal parameter which is given the attributes of label variable, the first word points to the address of the first word of the argument passed whereas the second word contains '1st add' of the argument.



Till now our discussion was limited to scalar variables.
We will now consider Arrays.

Storage representation:



($3n+1$ words where n
is the number of
dimensions.)

Thus it can be seen that array consists of three disjoint sets of storage locations. Every array is allocated a header from the first order storage (one word for all types of arrays except for label arrays with list which have two word header. Second word stores the list address), whose address is filled in the symbol table. Header word points to the dope vector and the actual area. Dope vector consists of information about bounds etc. and is assigned area from the first order storage area

(dope vector region). The actual area for the elements of the array is assigned from the second order storage or the run time stack.

There is no difference between the representation of a formal array and that of a normal array.

The format of array elements is exactly like the format of scalar variables for all types; except for the character variable for an element of a character array, there is no header word. Therefore, if an element is transmitted as an argument, the address supplied would directly point to the character element whereas for the character scalar, it would point to the header word and hence one level of indirection would have to be done to get the address of the actual character string element. To distinguish between these two cases the sign of header for a character scalar is made negative.

D-2 Major Types and Minor Types:

All the symbols which can be given an attribute in PL 7044, have been classified under two main categories:

1. Major Types
2. Minor Types

Major Types:

Following is the list of major types :

<u>Major Type</u>	<u>Description</u>
1	Normal Array
2	Formal Array
3	Static Array
4	Not applicable (Defined Array)
5	Normal Scalar
6	Formal Scalar
7	Static Scalar
8	Not applicable (Defined Scalar)
9	PROCEDURE without entry list
10	PROCEDURE with entry list
11	PROCEDURE BUILTIN
12	PROCEDURE MAP
13	PROCEDURE GENERIC
14	PROCEDURE Open ended
15	Pseudo Variable
16	Normal major/minor structure
17	Formal major/minor structure
18	Label constants
19	Formal entry without entry list
20	Formal entry with entry list
21	File constant
22	Formal file
23	PROCEDURE MAP with entry list
24	PROCEDURE BUILTIN where arrays are allowed.

The variables, corresponding to major types 1 to 8 are further divided into minor types. The list of minor types is as follows:

<u>Minor Type</u>	<u>Description</u>
1	Integer
2	Floating point
3	Complex
4	Not applicable (integer numeric picture)
5	Not applicable (floating point numeric picture)
6	Bit string- Fixed length
7	Bit string - varying length
8	Character string - fixed length
9	Character string - varying length
10	Not applicable (non numeric picture)
11	Label without label list
12	Label with label list

D-3: Symbol Table Representation of Various Major Types:

Before describing about individual major types, it would be appropriate to look at the typical symbol table cell shown on the next page.

Nomenclature:

LSTCK: Address of the last cell in stack Index table offset.
 INDXOF: Index table offset.
 LSTSNM: Address of the last cell in stack with same name

ff1	ff2	LSTCK	fst chr	mutt af	INDXOF*
S-element		LSTSNM	Line flag	Int'l flag	assclad/intlstad
		NXTSTR			LSRSNM
		NXSTLV*	binflag		COVER*
		address/ serial number	NDIMEN	MJRTYP	BLKNO
n	←----- do no. ----->	p	STRNO	STRLVL	MNRTYP

SYMBOL TABLE CELL

Note: Items with asterisk are stored in 2's complement notation.

NXTSTR: Address of the cell next in structure.
 LSRSNM: Address of the last cell in stack with same name in the structure.
 NXSTLV: Address of the cell representing next sister node.
 COVER: Address of the covering item cell.
 NDIMEN: Number of dimensions for variables or number of parameters for an entry name, zero if not applicable.
 MJRTYP: Major type
 BLKNO: Block number
 STRNO: Structure number
 STRIVL: Structure level
 MNRTYP: Minor type (for entry constants it stores the minor type of result returned).
 address/: Address of first order storage for variables.
 serial no Address of PHW for entry constants. Label serial number for label constants. File serial number for file constant.
 assclad : Associated cell address for built in procedures, and procedures with entry list.
 intlstad: Initial list address for structure elements
 n,p: Precision information for arithmetic variables.
 Primary and secondary types of file for a file constant
 dono.: DO number for a label constant.
Flags:
 ff1: Formal flag -1, '1' if explicit declaration in DECLARE statement occurs, '0' otherwise.

ff2: Formal flag-2, '1' for formal parameter

fstchr: first character flag, '1' if initial character of the name is I-N, '0' otherwise.

multdf: multiple definition flag,
set to '1' if a declaration at level 0 or 1 already exists in a block and another declaration is tried in the same block having same name.

likeflag: '1' if this structure is expanded by LIKE attribute

intlflag: Initial flag,
'1' if this variable has been initialised by INITIAL attribute in DECLARE statement.

s-element: Structure element flag,
'1' if it is a base element of a structure

bin flag: Binary flag,
'1' if the arithmetic variable is given a base of BINARY.

Note: Fields and flags which are irrelevant to a major type, contain zeros.


Symbol table representation of variables, file constants and label constants has nothing extra than what is shown in the symbol table cell.

Symbol table representation of entry constants having entry list and generic-procedure would be described below because they involve additional cells. Format for a builtin procedure is described in Appendix H.

Symbol table representation of entry constants having entry list:

For an entry constant having entry list, there are two cells-main and associated, in the symbol table. The main cell is as usual and points to the associated cell (assclad). The associated cell is used to store the type information of various arguments.

Format of associated cell:

NARG		
ARG-1		
ARG-2		
ARG-3		
ARG-4		
ARG-5		
ARG-6		
ARG-7		
ARG-8		
ARG-9		
ARG-10		
ARG-11		
ARG-12		
ARG-13		
ARG-14		
ARG-15		

where NARG is number of arguments in the entry list. For the n-th parameter, ARG-n field consists of

pmy-type	sndy-type
----------	-----------

where "pmy-type" is one of the following:

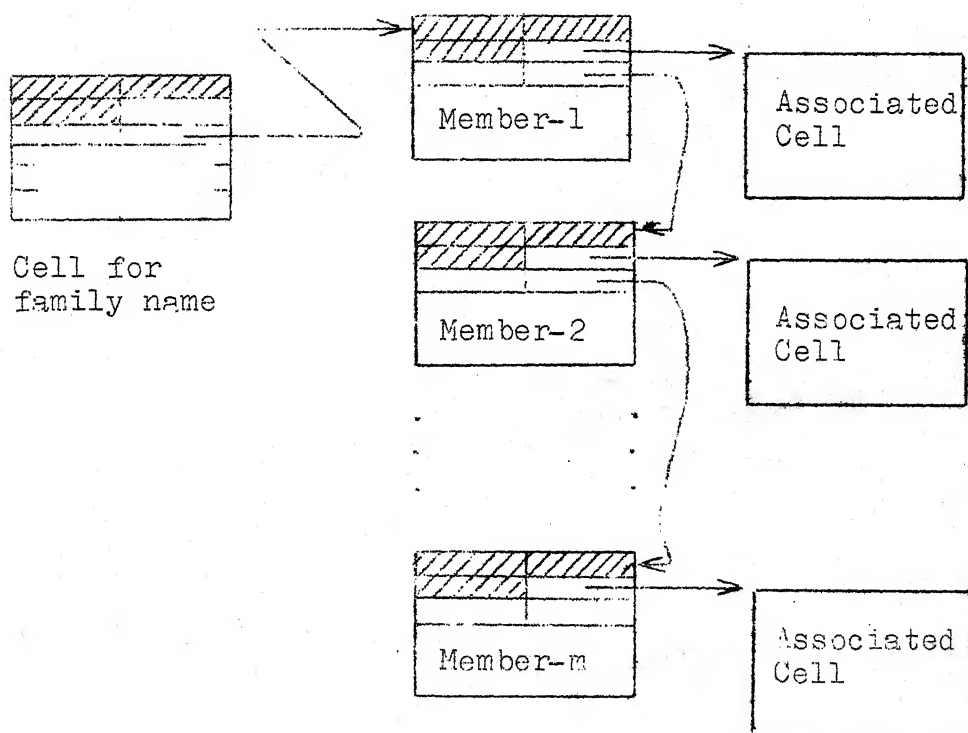
pmy-typeParameter descriptor

0	:	Star/Null
i,(i=1-5):	:	i-dimensional array
6	:	Structure
7	:	Scalar
8	:	Entry
9	:	File

"sndy-type" is one of the 12 minor types described earlier. For an n-argument entry list, remaining $15-n$ ($n \leq 15$) ARG-fields would contain zero.

Because of the limitation put by the size of the cell, more than 15 parameters can not be specified in an entry list.

Symbol table representation for GENERIC family:



(The hatched portion shows the usual pointers).

Each member of the family is represented exactly like an entry constant with entry list. However, a chain is formed starting from the GENERIC cell which links all the members. LSRSNM is used to store this pointer. The chain is terminated by '0' in next member address.

APPENDIX E

A program organised to explain the concepts of scope of declaration, extension of scope and use of BUILTIN attribute:

```
EXT1:  PROCEDURE OPTIONS (MAIN);
      L1: DECLARE ((A,B) EXTERNAL, C,SQRT(10)) FIXED (6,3),
      S1: GET LIST (A,B, SQRT);
      L2: CALL INT1;
      L3: B=EXT2(SQRT(A));
      L4: CALL INT2;
      S2: Statement
      INT1: PROCEDURE;
          L5: DECLARE (B,C(10)) FIXED (6,3), D STATIC;
          S3: C = B+D+2*SQRT+E;
              END INT1;
      INT2: PROCEDURE;
          L6: DECLARE C FIXED (6,3), A FLOAT(8),SQRT BUILTIN;
          S4: C=B+0.7*SQRT(C);
              END INT2;
          S5: Statement
      END EXT1;
EXT2:  PROCEDURE (X) RECURSIVE;
      L7: DECLARE (A,B) FIXED (6,3) EXTERNAL,
              C FLOAT (4) INITIAL (0), X FIXED (5,0);
      S6: IF X-2* X/2=0 THEN CALL INT3;
      S7: GET LIST (B);
      INT3: PROCEDURE;
          L8: DECLARE (A) FLOAT (4);
          S8: C = B+A
              END INT3;
      S9: IF X=1 THEN RETURN (1.);
          ELSE RETURN (C+X*EXT2 (X-1));
      END EXT2;
```

Note: " Statement." refers to a PL-7044 statement.

A table would be made to show the attributes and scopes of various variables.

Statement Label	Name	Attributes	Scope
EXT1	EXT1	ENTRY,EXTERNAL	Entire program
L1	A	FIXED,EXTERNAL	All of EXT1 except INT2 and All of EXT2 except
L1	B	FIXED,EXTERNAL	All of EXT1 except INT1 and All of EXT2
L1	C	FIXED,INTERNAL	All of EXT1 except INT1 and INT2
L1	SQRT	FIXED,INTERNAL, Array	All of EXT1 except INT2
INT1	INT1	ENTRY,INTERNAL	All of EXT1
L5	B	FIXED, INTERNAL	All of INT1
L5	C	FIXED,INTERNAL, Array	All of INT1
S3	E	Default	All of EXT1
INT2	INT2	ENTRY,INTERNAL	All of EXT1
L6	C	FIXED,INTERNAL	All of INT2
L6	A	FLOAT,INTERNAL	All of INT2
L7	SQRT	BUILTIN	All of INT2
EXT2	EXT2	ENTRY,EXTERNAL	Entire program
L7	A,B	FIXED,EXTERNAL	Same A and B as declared in L1
L7	C	FLOAT,INTERNAL	All of EXT2
L7	X	FIXED, INTERNAL (parameter)	All of EXT2
INT3	INT3	ENTRY,INTERNAL	All of EXT2
L8	A	FLOAT,INTERNAL	All of INT3

Examination of this program will reveal the following facts:

- i) There are three separate locations identified by same name, A. The first of these is allocated by the declaration in statement L1 as a fixed point decimal number with the EXTERNAL attribute. As such it is also given the STATIC attribute by default. The second location name 'A', declared at L6, refers to a 8 digit floating point decimal number known to only procedure INT2.

The third is a floating point decimal number with the internal attribute, declared at L8, which remains allocated during the invocation of procedure INT3.

The A declared in L7 is the same as A declared in L1.

- ii) There are two different locations with name B. The first is declared in statement L1, whereas the second is declared in statement L5 and is known only to procedure INT1.

Though both the B's are fixed point decimal numbers with same precision, the fact that second B is declared, explicitly, in an internal procedure results in a separate allocation and release upon termination of that procedure.

- iii) The program assigns four different locations using the name C, one of them being the header of an array.

- iv) The name SQRT is having two different meanings in this program. SQRT is declared in L1 as a one dimensional array of floating point decimal numbers with single precision. It's use in statement L3 is as a floating point variable.

In statement S3 it is being used in an aggregate expression. The use of SQRT in statement S4 is as a built in procedure and it represents a call to the builtin procedure with name SQRT. Thus use of SQRT in external procedure L1 as an array does not bar its use in an internal procedure as a builtin name, if declared appropriately.

- v) The procedure EXT2 has been given RECURSIVE attribute and it is being called from within itself at statement S9. In such cases a rule for terminating the recursion is also to be given, otherwise it will form an infinite loop. At each call of EXT2, a value of B is read in from input media but only the value which is existing when a call is made to INT3, will be transmitted to INT3.

APPENDIX F

First Order Storage Assignment For Various Blocks of a Program:

To illustrate the first order storage assignment for various blocks of a program, program of Appendix E would be considered once again.

a. Block Predecessor Table:

<u>BIPDTB</u> (at compile time)	<u>b.no.</u>	<u>name</u>	<u>Run time coding</u>
			BIPDTB EQU *
0	0	imaginary	OCT 0
0	1	EXT1	OCT 0
1	2	INT1	OCT 1
1	3	INT2	OCT 1
0	4	EXT2	OCT 0
4	5	INT3	OCT 4

b. Block Storage Table:

		<u>STGTAB EQU *</u>
0	imaginary	PZE BE.00,,BS.00
1	EXT1	PZE BE.01,,BS.01
2	INT1	PZE BE.02,,BS.02
3	INT2	PZE BE.03,,BS.03
4	EXT2	PZE BE.04,,BS.04
5	INT3	PZE BE.05,,BS.05

C. Labels Generated:

```

AS.00 EQU BS.00+SEA(00)
TS.00 EQU AS.00+DPVC(00)
BE.00 EQU TS.00+TA(00)

BS.01 EQU BE.00
AS.01 EQU BS.01+SEA(01)
TS.01 EQU AS.01+DPVC(01)
BE.01 EQU TS.01+TA(01)

BS.02 EQU BE.01
AS.02 EQU BS.02+SEA(02)
TS.02 EQU AS.02+DPVC(02)
BE.02 EQU TS.02+TA(02)

BS.03 EQU BE.01
AS.03 EQU BS.03+SEA(03)
TS.03 EQU AS.03+DPVC(03)
BE.03 EQU TS.03+TA(03)

BS.04 EQU BE.00
AS.04 EQU BS.04+SEA(04)
TS.04 EQU AS.04+DPVC(04)
BE.04 EQU TS.04+TA(04)

BS.05 EQU BE.04
AS.05 EQU BS.05+SEA(05)
TS.05 EQU AS.05+DPVC(05)
BE.05 EQU TS.05+TA(05)

BS.00 EQU *
      BSS MXFSAR

```

where,

~~MXFSAR~~ represents maximum first order storage area required

SEA(bn) represents the scalar elements area for block 'bn'

DPVC(bn) represents the dope vector area for block 'bn'

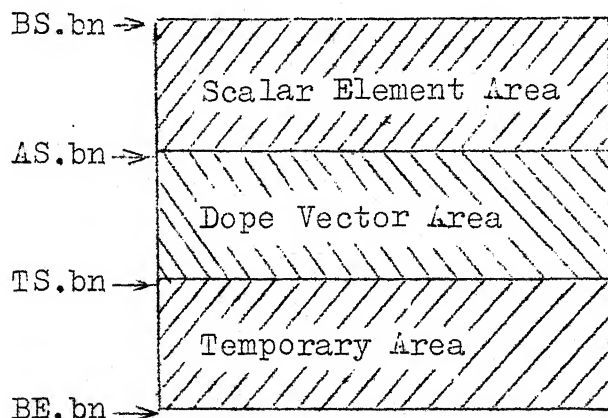
TA(bn) represents the temporary area for block 'bn'

Some interesting conclusions can be drawn from the above exercise.

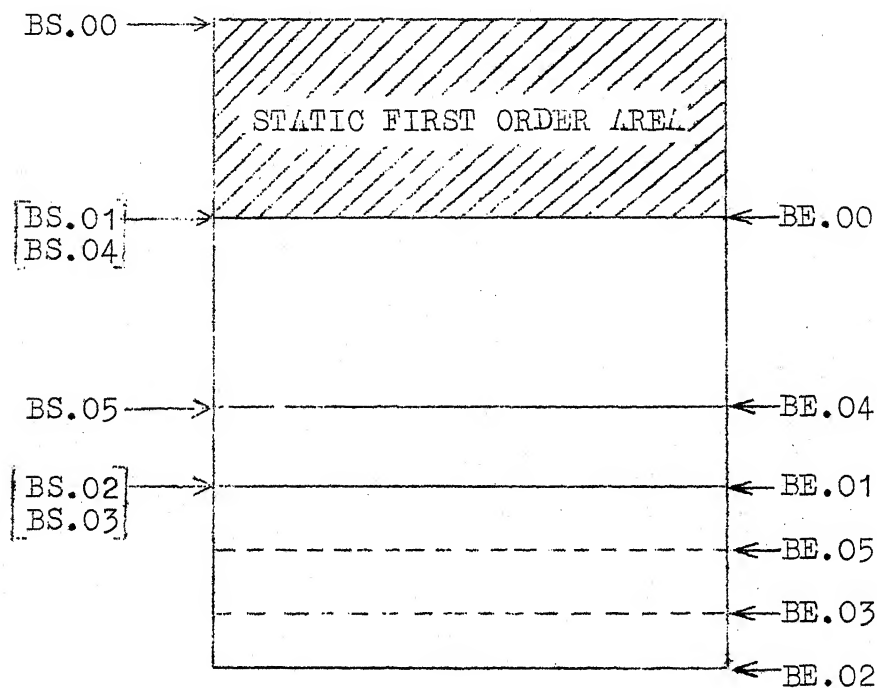
1) Starting addresses of blocks 2(INT1) and 3(INT2) and blocks 1(EXT1) and 4(EXT2) are same. This shows that they share the same first order storage area. This is because of parallelism in their definition. In fact any two blocks having same predecessor would share the same first order area.

ii) Storage assigned to block '0' (imaginary) corresponds to the STATIC variables.

d. Segments of First Order Storage for a Block:



e. Schematic diagram of I order storage assignment for the program considered:



APPENDIX G

G-1 Attributes of Files:

The following shows the attributes that may be assigned to a file as part of a DECLARE statement.

<u>options</u>	<u>Default</u>
INPUT, OUTPUT, UPDATE, PRINT	INPUT
STREAM, RECORD	STREAM
BUFFERED, UNBUFFERED	BUFFERED
SEQUENTIAL	

The file-attributes set is completed as follows:

<u>If Declaration Specifies</u>	<u>PL/7044 will also Assign</u>
UPDATE	RECORD
SEQUENTIAL	RECORD
PRINT	OUTPUT, STREAM
BUFFERED or UNBUFFERED	RECORD, SEQUENTIAL

G-2 Default Attributes:

Storage and scope:

All level one variables: AUTOMATIC is the default for INTERNAL level-one variables; STATIC is the default for EXTERNAL level-one variables. If neither the storage class nor the scope is specified, AUTOMATIC and INTERNAL are assumed.

Arithmetic data: If no attribute is specified for a variable, default attributes are assigned depending upon the first character (or the only character) of the variables. Integer (27 bits) is taken if the first character happens to be I-W. Floating point of single precision is assumed for all the remaining.

String data: If no length specification is specified for a string variable, it is taken as 1.

APPENDIX H

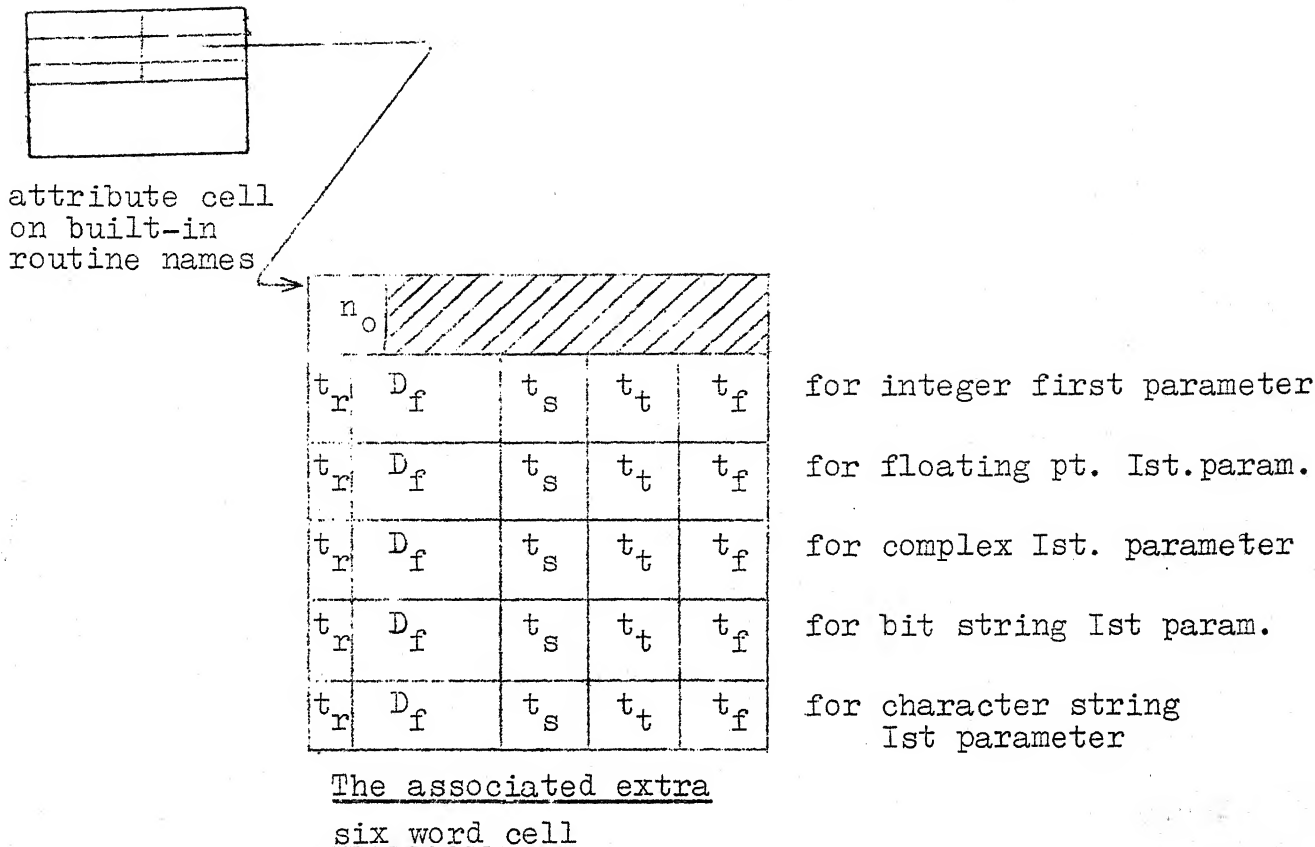
Builtin Procedures and Pseudo Variables:

Because of the generic nature of all the three types of built-in routines (simple built-in, array handling built-in and pseudo variable) of PL/I, symbol table associates an extra six word cell with the attribute cell of builtin name, for holding the description of the generic family members. Selection among these generic members is performed depending on the type of the first parameter of the built-in routine reference. The second, the third, the fourth, the fifth and the sixth word of the associated cell hold the description of generic member for integer, floating point, complex, bit string and character string type of first parameter. Figure APNG-1 illustrates this organisation.

Coding of the type specification (i.e. of t_r , t_s , t_t and t_f) for generic family members varies among the three types of built-in routines. For each type of the built-in routine, the coding of t_r , t_s and t_f is described below.

a) Coding of t_r , t_s , t_t and t_f for built-in routine:

- $t_r=1$ when the returned result is of integer type.
- $=2$ when the returned result is of floating point type.
- $=3$ when the returned result is of complex type.
- $=4$ when the returned result is of bit string (always non-varying) type.
- $=5$ when the returned result is of character string (always non-varying) type.



where,

n_o = number of parameters associated with this built-in routine.

t_r = type of the result returned by a member of the generic family.

D_f = Identification number of the entry point for a member of the generic family.

t_s = type in which second parameter is expected.

t_t = type in which the third parameter is expected.

t_f = type in which the fourth parameter is expected

Note: No built-in routines can have more than four parameters. But in case there are more than four parameters supplied in any built-in routine call, all parameters from fifth onwards will be converted, in accordance with the specification for the fourth parameter.

- $t_s, t_t, t_f = 0$ when any type of argument is allowed in that parameter position.
- = 1 when only integer argument is allowed in that parameter position.
 - = 2 when only floating point argument is allowed in that parameter position.
 - = 3 when only complex argument can be passed on as parameter.
 - = 4 when only bit string (any of varying or non-varying type) can be passed on as parameter.
 - = 5 when only character string (any of varying or non-varying type) can be passed on as parameter.
 - = 6 when only literal integer can appear in that parameter position.
 - = 7 Either of bit string or character string type of argument can be passed on as parameter. For any other type of argument conversion to character string, is implied.

b) Coding of t_r, t_s, t_t and t_f for array handling built-in routine:

Coding of t_r for array handling built-in routines is identical to that for built in routines of case (a).

- $t_s, t_t, t_f = 0$ when any type of argument is allowed in that parameter position.
- = 1,2,3,4,5 for legal appearance of only scalar integer, floating point, complex, bit string and character string arguments in the parameter position.

= 6,7,8,9,10 for legal appearance of only integer, floating point, complex, bit string and character string arrays in that parameter position.

- c) Organisation of attribute cell for a pseudo variable and coding of t_r, t_s, t_t and t_f for pseudo variable generic family members:

Since there exists, always, a built-in function having the same name as pseudo variable, the attribute cell for any pseudo variable name contains a pointer to the attribute cell of the built-in function of same name. Buffer handler chooses (depending on the context) the correct alternative and for built-in function of the same name it follows that pointer to get the corresponding attribute cell address. Organisation of the symbol table entry, corresponding to a pseudo variable name is illustrated in Fig. APNG-2.

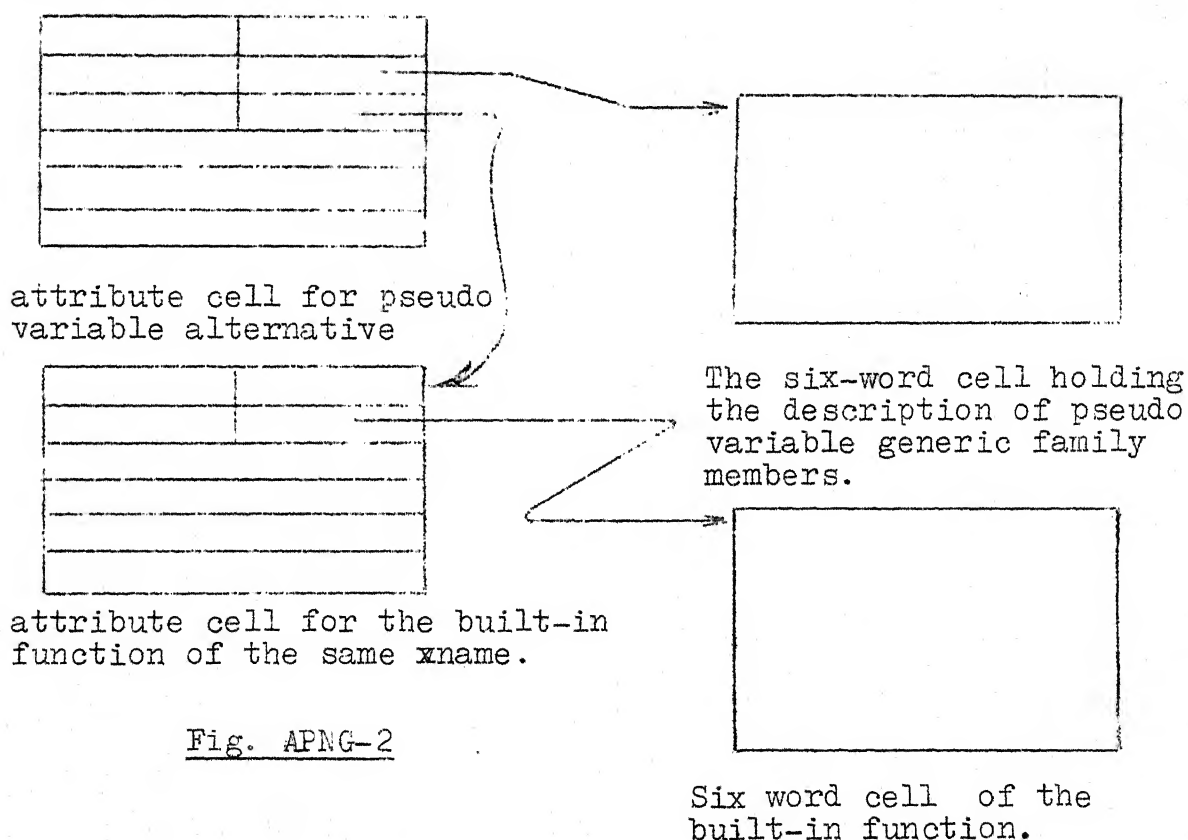


Fig. APNG-2

Coding of t_r , D_f , t_s , t_t and t_f for built in function alternative is identical to that for any other built, in function (case a). For pseudo variable alternative,

t_r = 1 for pseudo variable requiring complex result of the right hand side of '=' expression.
 = 2 for pseudo variable requiring bit string (varying or non-varying) result of right hand side of '=' expression
 = 3 for pseudo variable requiring character string (varying or non-varying) result of right hand side of '=' expression.

t_s, t_t, t_f = 0 when every operand is allowed in that parameter position.
 = 1,2,3,4,5 when integer, floating point, complex, bit string (varying or non-varying) and character string (varying or non-varying) variable (and not expression) is allowed in that parameter position.
 = 6,7,8,9,10 when integer, floating point, complex, bit string (varying or non-varying) and character string (varying or non-varying) operand (either variable or expression) is allowed in that parameter position.

APPENDIX I

Statements, Attributes and Restrictions over PL/I:

I-1 Statements:

Following are the alphabetic lists of statements allowed and statements not allowed in PL 7044.

ALLOWED

BEGIN
CALL
CLOSE
DECLARE
DISPLAY
DO
ELSE
END
ENTRY
EXIT
FORMAT
GET
GOTO
GO TO
HALT
IF
OPEN
PROCEDURE
PUT
READ
RETURN
STOP
WRITE

NOT ALLOWED

ALLOCATE
DEFAULT
DELAY
DELETE
FETCH
FLOW
FREE
LOCATE
NOFLOW
ON
RELEASE
REVERT
REWRITE
SIGNAL
UNLOCK
WAIT

I-2: Attributes:

Attributes have been divided in four major classes. Each class is further sub-divided in various minor classes.

1. Storage and Scope
 - A. Storage
 - B. Scope
2. Data
 - A. Arithmetic
 - B. String
 - C. Label
 - D. Entry
 - E. File
 - F. Task and program control
3. Miscellaneous
4. No-action

The digits and the letters (if any) appended to attributes in the following alphabetic lists of attributes allowed and attributes not allowed, refer to the major class and minor class respectively.

ALLOWED

ABNORMAL (4)

ALIGNED (4)

AUTOMATIC (1-A)

BINARY (2-A)

BIT (2-B)

NOT ALLOWED

AREA (2-F)

BACKWARDS (2-E)

BASED (1-A)

CELL (1-A)

CONTROLLED (1-A)

BUFFERED (2-E)	DEFINED (3)
BUILTIN (2-D)	DIRECT (2-E)
CHARACTER (2-B)	ENVIRONMENT (2-E)
COMPLEX (2-A)	EVENT (2-F)
DECIMAL (2-A)	EXCLUSIVE (2-E)
DIMENSION (3)	KEYED (2-E)
ENTRY (2-D)	OFFSET (2-F)
EXTERNAL (1-B)	POINTER (2-F)
FILE (2-E)	PICTURE (2-B)
FIXED (2-A)	PACKED (4)
FLOAT (2-A)	POSITION (3)
GENERIC (2-D)	TASK (2-F)
INITIAL (3)	
INPUT (2-E)	
INTERNAL (1-B)	
IRREDUCIBLE (2-D)	
LABEL (2-C)	
Length (2-B)	
LIKE (3)	
MAP (2-D)	
NORMAL (4)	
OUTPUT (2-E)	
Precision (2-A)	
PRINT (2-E)	
REAL (2-A)	

RECORD (2-E)

Record-size (2-E)

REDUCIBLE (2-D)

RETURNS (2-D)

SECONDARY (1-A)

SEQUENTIAL (2-E)

SETS (2-D)

STATIC (1-A)

STREAM (2-E)

UNBUFFERED (2-E)

UPDATE (2-E)

USES (2-D)

VARYING (2-B)

I.3: Restrictions:

a) Logical Restrictions:

1. Arrays of structures are not allowed.
2. Minor/major structure name should not be used in a GET/PUT DATA statement.
3. Aggregate expression can not be used as an argument in a procedure reference.
4. Recursive function reference should not be made from inside dimension attribute, length attribute, iteration factor or format statement.

5. All DECLARE statements for a block should be placed immediately after the header statement (PROCEDURE/BEGIN). All specifications for parameters of a secondary entry point should be put immediately after the ENTRY statement. No declaration can be made after an ENTRY statement.
6. DO loops can not be used within a GET/PUT DATA statement.
7. Character/Bit strings can not be used within a list or edit or data directed I/O command.
8. A data list must always be specified with a GET DATA command.
9. When a file name is to be transmitted as an argument in a procedure reference, it must either be opened by a OPEN statement or must be used in I/O command lexicographically before its occurrence as argument.
10. A procedure block can not be defined inside a DO group.

b) Storage Restrictions:

1. A statement can be punched anywhere between column 1 to column 72 (both inclusive) on a 80-column card. Last eight columns may be used for identification.
2. Maximum length of identifiers and character string constant is 30 characters.
3. Maximum length of a bit string constant is 36 bits.
4. Largest absolute value of an integer is $2^{27}-1$ (163708927) (27 bits).

5. A floating point constant should satisfy the relationship $10^{-38} \leq |\text{fp. constant}| \leq 10^{38}$.
6. More than 62 blocks can not be used in the program.
7. More than 63 DO-groups (excluding those used in I/O statements) can not be used.
8. More than 63 structures can not be used.
9. More than 4095 IF statements can not be used.
10. More than 8 labels can not be prefixed to a statement.
11. More than five dimensions in an array can not be used.
12. More than fifteen parameters can not be used while defining a PROCEDURE.
13. More than five structures can not take part in an aggregate expression. Further none of the structures taking part in the aggregate expression should have more than 25 base elements, Structures taking part in a structure expression with BY NAME option specified, should not have more than 10 items at level L+1 under an item at level L.
14. Maximum number of files corresponds to number of utility units available in the installation.
15. A statement can be 500 output-units long at the most. The table for lexical-units and number of output-units is as follows:

<u>Lexical-unit</u>	<u>No. of Output Units</u>
Operators	1
Integers floating point constant; identifier and key words	2
Complex and bit string constants	3
Character string constants	$\lceil n/6 \rceil + 2$

where, n is the length of character string constant.

APPENDIX J

J-1 Program Listing:

Program listing is generated during the first edition. Besides the card-image, it also carries three identification numbers viz. statement number, card number and the block number.

Statement number is used in second edition to refer to the corresponding statement if an error is detected in compilation.

Block number indicates the pairing of the 'END' and the block header statements (PROCEDURE/BEGIN).

The first pass processor prints the type of the END (as to whether closing a DO-group, BEGIN-block or PROCEDURE-block) also. In case of multiple closure besides one such message for each group/block closed, it also prints the message 'MULTIPLE CLOSURE ENCOUNTERED'.

The information about generation of a dummy ELSE is also conveyed to the programmer. The message to this effect appears misplaced on the listing. It should actually appear before the statement detecting the missing ELSE, whereas it appears after that statement. It could have been taken care of but at the cost of increased complexity, which is not warranted if one is familiar to this convention.

The above three facilities help the programmer to keep track of his logic. This is especially helpful in cases like shuffling of cards etc.

At the successful completion of an edition, a message is printed to this effect which shows the progress made by the compiler.

J-2 Errors and the Action Taken:

Complete error message is not printed for the compile time errors. A 6 character mnemonic (3 alphabetic letters followed by 2 digits, 'XXX-nn') uniquely represents an 'error'. However for errors detected at run time, complete error message is printed.

All the errors detected in the first edition appear interleaved in the program listing. An error detected during the scanning of a statement appears immediately after the card-image of the card on which the error has been detected. However, if the error is detected after the scanning phase is over, the error code follows the card-image on which the statement ends.

At the end of an edition, if it is found that an error has been detected in this edition, all further editions are deleted and the job is terminated. When an error is detected in a statement, it is immediately skipped and no further analysis is done for that statement.

The error messages of the second pass carry the statement number of the statement in error so that proper referencing can be done, since they appear after the program listing.

APPENDIX K

Format of the File Status blocks associated with each stream file:

All file status blocks are 8 words in size.

(a) When file is an input stream file,

PFXI	
	RECNO
	CHRPTR
	WRDPTR
	SIGN
	GETCH2
	GETCH1
PFX2	

where,

RECNO: Record Number = no. of records processed from this file.

CHRPTR: Character pointer - gives the number of character still to be processed in the word pointed to by GETCH1.

WRDPTR: No. of words still to be processed in the current data record.

GETCH2: -ve if a new record is to be read.

+ve if a new record is not to be read.

SIGN: -ve for system input file +ve for others.

GETCH1 = Address of the word in the current record being processed.

PFX1 = PZE for stream print files
 = PON for stream input
 = MZE for record input
 = MON for record output

PFX2 = PZE if file has not been used
 = MZE if file has been used

b) When the file is an output stream file:

PFX1			Addr1
LINE NO			
BU22.			
CHRPTR			
WRDPTR			
PFX2			Addr2

where,

Addr1 = link address of the next file status block
 = 0 if this is the last link
 ≠ 0 if it is not the last link

Addr2 = address of file control block of the file
 corresponding to this file status block.

LINENO = number of lines or records written in this file.

BU22. = pending word in this file.

· CHRPTR = number of character available in the pending word.

WRDPTR = address of word in buffer in complemented from where the next output word is to be placed.

PFX1 and PFX2 have same meanings as in (a).

APPENDIX L

L-1: Buffer handler output for an Assignment Statement:

The coding of the assignment statement $A(I,*,J,*),C=$
 $I+S+J+T+C1+1+2.0+C2+2+2I+A(*,K1,*,K2)+A(I,*,J,*)+(B*,K1,*)+C(*,*)+$
 $D(1,2);$ where I, J, D, K1 and K2 have been declared as integer,
 B, S and T have been declared as floating point and A, C1 and C2
 have been declared to be complex, by buffer handler is given below:

Notation:

catpx in the buffer handler output stands for the complemented address of the attribute cell of the Identifier X.

<u>Buffer handler output in Octal</u>	<u>Comment</u>
0 catpa 0 00032	Operand type of 32_8 denotes the following of array subscripts for address calculation.
0 catpi 0 00001	Operand type of 1 stands for scalar identifier for which normal (and not indirect) address calculation is to be done.
4 00004 0 00044	Operator ',' separating array subscripts.
0 catpj 0 00001	
4 00004 0 00044	
4 00004 0 00053	Operator ')' denoting end of array subscripts in an array cross-section reference.
0 00000 0 00002	Number of non '*' subscripts.
0 20400 0 00002	Representative word of array cross section.
4 00004 0 00021	First ',' separating multiple assignment arguments.
0 catpc 0 00033	Operand type of 33_8 denotes array without any subscript list reference.

4 00004 0 00030	Subsequent ',' separating multiple assignment arguments.
4 00014 0000026	Representation of assignment '='.
0 00000 0 00002	Number of multiple assignment arguments.
0 catpi 0 00001	
4 00011 0 00006	Binary '+' operator.
0 catps 0 00002	Operand type of 2 denotes scalar operand for which indirect addressing scheme is to be adopted.
4 00011 0 00007	Binary '-' operator.
0 catpj 0 00001	
4 00011 0 00006	
0 catpt 0 00001	
4 00011 0 00006	
0 catpc ₁ 0 00001	
4 00011 0 00006	
0 00000 0 00003	Operand type of 3 is for literal integer.
0 00000 0 00001	Value of the literal.
4 00011 0 00006	
0 00000 0 00004	Operand type of 4 is for floating point literal.
2 02400 0 00000	Value of the literal.
4 00011 0 00006	
0 00000 0 00005	Operand type of 5 is for complex literal.
2 02400 0 00000	Real part of the literal.
2 02400 0 00000	Imaginary part of the literal
4 00011 0 00006	
0 catpa 0 00032	Starting of coding of A(*,K1,*,K2).
0 catpk ₁ 0 00001	
4 00004 0 00044	
0 catpk ₂ 0 00001	
4 00004 0 00044	
4 00004 0 00053	') ' denoting end of subscript list for array cross section reference.
0 00000 0 00002	
0 10300 0 00002	

4 00011 0 00006

0 catpb 0 00032

Starting of coding of $B(*, K2, *)$

0 catpk,0 00001

4 00004 0 00044

4 00004 0 00053

0 00000 0 00001

0 10300 0 00002

4 00011 0 00006

0 catpd 0 00032

Starting of coding for $D(1,2)$.

0 00000 0 00003

0 00000 0 00000 1

4 00004 0 00044

0 00000 0 00003

0 00000 0 00002

4 00004 0 00044

4 00004. 0 00051

') ' denoting end of subscript list for array element reference.

4 00001 0 00027

Representation of operator devoting end-of-...
expression. This will be inserted by the calling
routine before calling expression processor.

L-2: Expression Processor Output (Before Sorting):

For buffer handler's coding of the assignment statement of Appendix-L1, the unsorted output of AExpression Processor in output buffer and the corresponding entries made in the summary record area are shown below:

Notations:

(1) Locations of output buffer are addressed symbolically by the symbols attached with the locations. Putting a star on top of those symbols denote 2's complement of those location addresses. (e.g. Sadd1* means 2's complement of the address of location Sadd1).

(ii) Block number and offset of addressing for an identifier X is represented as bx and offsx respectively.

Contents of Output Buffer

<u>Symbolic address</u>	<u>Content of that location in octal</u>	<u>Comment</u>
Sadd1	0 00000 0 00006	Record header for the recursive address calculation output record of A(I,*,J,*)
	0 00101 0 00113	Opcode 113 ₈ is for generating recursive address calculation output. 01001 ₈ in decrement is the label where address of array elements will be filled in.
	0 offsa 0 001ba	Address of array header A(in expression processor's output form).
	0 20400 0 00002	Cross section representative word of A(I,*,J,*).
	0 00000 0 00002	Number of non '*' subscripts.
	0 offsi 0 101bi	Subscript I
	0 offsj 0 101bj	Subscript J
Sadd2	0 00000 0 00004	Output for recursive address calculation of c.
	0 00102 0 00113	
	0 offsc 0 001bc	
	0 10203 0 40502	
	0 00000 0 00000	
Sadd3	0 00000 0 00006	Output for recursive address calculation of A(*,K1,*,K2).
	0 00103 0 00113	
	0 offsa 0 001ba	
	0 10300 0 00002	
	0 00000 0 00002	
	0 offsk ₁ 0 101bk ₁	
	0 offsk ₂ 0 1016k ₂	
Sadd4	0 00000 0 00006	Output for recursive address calculation of A(I,*,J,*).
	0 00104 0 00113	
	0 offsa 0 001ba	
	0 20400 0 00002	
	0 00000 0 00002	
	0 offsi 0 101bi	
	0 offsj 0 101bj	

Saad5	0 00000 0 00005	
	0 00105 0 00113	
	0 offsb 0 001bb	Output for recursive address
	0 10300 0 00002	calculation of B(*,K2,*).
	0 00000 0 00001	
	0 offsk ₂ 0 101bk ₂	
Saad6	0 00000 0 00004	
	0 00106 0 00113	
	0 offsc 0 001bc	Output for recursive address
	0 10203 0 40502	calculation of C(*,*).
	0 00000 0 00000	
Sadd7	0 00000 0 00005	
	0 00107 0 00112	
	0 offsd 0 001bd	Output for address calculation
	0 00000 0 00002	of D(1,2)
	0 00014 0 10500	
	0 00015 0 10500	
Sadd8	0 00000 0 00032	Record header for scalar output.
	0 00001 0 00012	12 ₈ is opcode for integer addition.
		00001 ₈ is the temporary which will
		contain the result of this operation.
	0 offsi 0 101bi	
	0 offsj 0 101bj	Negative sign with this operand shows
		it is bared.
	0 00002 0 00012	
	0 00001 0 11300	Temporary 1
	0 00011 0 10500	Literal 1
	0 00003 0 00012	
	0 00002 0 11300	
	0 01007 0 10700	D(1,2)
	0 00004 0 00071	71 ₈ is opcode for converting integer
		operand into floating point.
	0 00003 0 11300	
	0 00005 0 00013	Opcode for floating point addition.
	0 00004 0 21300	
	0 offss 0 201bs	
	0 00006 0 00013	
	0 00005 0 21300	
	0 offst 0 202bt	Indirect addressing is involved
		with T.

	0 00007 0 00014	14 ₈ is opcode for floating point and complex addition.
	0 00006 0 21300	
	0 offsc ₁ 0 301bc ₁	
	0 00010 0 00015	15 ₈ is opcode for complex addition.
	0 00007 0 31300	
	0 offsc ₂ 0 301bc ₂	
	0 00011 0 00015	
	0 00010 0 31300	
	0 00013 0 30500	Literal 2+2I
Saad9	0 00000 0 00022	Record header for repetitive output.
	0 00012 0 00014	
	0 01005 0 20700	B(*,K2,*)
	0 00011 0 31300	
	0 00013 0 00015	
	0 00012 0 31300	
	0 01003 0 30700	A(*,K1,*,K2)
	0 00014 0 00015	
	0 00013 0 31300	
	0 01004 0 30700	A(I,*,J,*)
	0 00015 0 00015	
	0 00014 0 31300	
	0 01006 0 30700	C(*,*)
	0 00000 0 00056	56 ₈ is opcode for two word complex assignment.
	0 01002 0 30700	C
	0 00015 0 31300	
	0 00000 0 00056	
	0 01001 0 30700	A(I,*,J,*)
	0 00015 0 31300	
Sadd10	0 00000 0 00012	
	0 00010 0 00130	130 ₈ opcode for bound checking of arrays. 00010 ₈ denotes serial no. of the recursive address calculation.

	0 00000 0 00010	Number of words associated with this operator.
	0 offsa 0 001ba	
	0 20400 0 00002	
	0 offsc 0 001bc	
	0 10203 0 40502	
	0 offsa 0 001ba	
	0 10300 0 00002	
	0 offsb 0 001bb	
	0 10300 0 00002	
Sadd11	0 00000 0 00001	
	0 00000 0 00132	132 ₈ is opcode denoting starting of repetitive codes.
Sadd12	0 00000 0 00003	
	0 00010 0 00133	133 ₈ is opcode for closing of the repetitive loops.
	0 00000 0 00134	Opcode 134 ₈ denotes the end of expression processor output.

Entries in Summary Recorded Area

Content of summary
record area in octal

Comments

0 00002 0 00000	Constant entry for collecting all outputs for pseudo variable calls. Zero in the address field denotes existence of no such output.
0 00003 0 Sadd9*	Constant entry for collecting all repetitive output.
0 00004 0 00000	Constant entry for collecting all iSUB defined array repetitive address calculation output.
0 00011 0 Sadd8*	Constant entry for collecting all scalar output.
0 00006 0 Sadd1*	Entry for recursive address calculation of A(I,*,J,*).

0 00006 0 Sadd2*	For C.
0 00006 0 Sadd3*	For A(*,K1,*,K2).
0 00006 0 Sadd4*	For A(I,*,J,*).
0 00006 0 Sadd5*	For B(*,K2,*).
0 00006 0 Sadd6*	For C(*,*).
0 00012 0 Sadd7*	For D(1,2) address calculation.
0 00010 0 Sadd10*	For array bound check orders.
0 00005 0 Sadd11*	For output denoting the starting of repetitive codes.
0 00001 0 Sadd12*	For output denoting the closing of the repetitive loops.

L-3: Expression Processor Output (After Sorting):

On sorting of expression processor's output (as shown in Appendix L-2) by the sorting routine, Final output, as written in output file, is given below:

<u>Final sorted output in Octal</u>	<u>Comment</u>
7 77777 7 77777	These two words signify the necessity of calling expression processor's code generat- ing routine in third pass.
0 00000 0 00031	
0 01007 0 00112	Output for D(1,2).
0 offsd 0 001bd	
0 00000 0 00002	
0 00014 0 10500	
0 00015 0 10500	
0 00001 0 00012	Scalar output
0 offsi 0 101bi	
4 offsj 0 101bj	
0 00002 0 00012	
0 00001 0 11300	
0 00011 0 10500	
0 00003 0 00012	
0 00002 0 11300	
0 01007 0 10700	
0 00004 0 00071	
0 00003 0 11300	
0 00005 0 00013	
0 00004 0 21300	

0 offst 0 202bt
 0 00007 0 00014
 0 00006 0 21300
 0 offsc₁ 0 301bc₁

Scalar output

0 00001 0 00015
 0 00007 0 31300
 0 offsc₂ 0 301bc₂

0 00011 0 00015
 0 00010 0 31300
 0 00013 0 30500

0 00010 0 00130

0 00000 0 00010

0 offsa 0 001ba

0 20400 0 00002

0 offsc 0 001bc

0 10203 0 40502

0 offsa 0 001ba

0 10300 0 00002

0 offsb 0 001bb

0 10300 0 00002

All output for the bound checking of the arrays.

0 00101 0 00113

0 offsa 0 001ba

0 20400 0 00002

0 00000 0 00002

0 offsi 0 101bi

0 offsj 0 101bj

Output: for recursive address calculation of A (I,*,J,*).

0 00102 0 00113

0 offsc 0 001bc

0 10203 0 40502

0 00000 0 00000

Output for recursive address calculation of C.

0 00103 0 00113

0 offsa 0 001ba

0 10300 0 00002

0 00000 0 00002

0 offsk₁ 0 101bk₁

0 offsk₂ 0 101bk₂

Output for recursive address calculation of A(*,K1,*,K2).

0 00104 0 00113

0 offsa 0 001ba

0 20400 0 00002

0 00000 0 00002

0 offsi 0 101bi

0 offsj 0 101bj

Output for recursive address calculation of A(I,*,J,*).

0 00105 0 00113

0 offsb 0 001ba

0 10300 0 00002

0 00000 0 00001

0 offsk₂ 0 101bk₂

Output for recursive address calculation of B(*,K2,*).

```

0 00106 0 00113
0 offsc 0 001bc
0 10203 0 40502
0 00000 0 00000

```

Output for recursive address calculation of C(*,*).

```

0 00000 0 00132

```

Output denoting starting of repetitive codes.

```

0 00012 0 00014
0 01005 0 20700
0 00011 0 31300
0 00013 0 00015
0 00012 0 31300
0 01003 0 30700
0 00014 0 00015
0 00013 0 31300
0 01004 0 30700
0 00015 0 00015
0 00014 0 31300
0 01006 0 30700
0 00000 0 00056
0 01001 0 30700
0 00015 0 31300
0 00000 0 00133
0 00000 0 00002
0 00000 0 00134

```

Output denoting end of repetitive codes.
 Number of repetitive loops to close.
 End of expression processor's output.

L-4: MAP CODE GENERATED:

For the final sorted expression processor's output (as shown in Appendix L-3), the third pass expression processor's code generating routine produces the following assembly language (MAP) card images.

Generated Card Images

Comment

LO0011 EQU	=0000000000001	
LO0012 EQU	=0202400000000	
LO0013 OCT	=202400000000,202400000000	
LO0014 EQU	=0000000000001	
LO0015 EQU	=0000000000002	
.		
.		
.		

These card images will be produced by second pass expression processor itself.

All card images produced, similarly, by expression processor while processing other statements in the programme will follow.

USE JUNK

Starting of codes for this particular assignment statement.

LO1001 PZE L00014
PZE L00015

Subscript list for D(1,2) address calculation.

USE CODE

TSX /ADDCAL,4
PZE L00001
PZE bd+offsd
STA L00107

Codes for address calculation of D(1,2).

CLA bi+offsi

SUB bj+offsj

ADD L00011

Literal 1 is added to the partial result of I+J.

LO0107 ADD **

Adding of D(1,2) to the partial result.

ORA =023300000000
FAD =023300000000

Converting partial result from integer to floating point.

FAD* bs+offss

FAD bt+offst

FAD L00012

Addition of literal 2.0 to the partial result.

FAD bc₁+offsc₁

STO Temp+0

Temp and Temp+1 are the temporary locations for holding partial result.

CLA bc₁+offsc₁+1

STO Temp+1

CLA Temp+0

FAD bc₂+offsc₂

STO Temp+0

CLA Temp+1

Addition of .02 to the partial result.

FAD bc₂+offsc₂+1

STO Temp+1

CLA Temp+0

FAD L00013

STO Temp+0

CLA Temp+1

Addition of literal 2+2I with the partial result.

FAD L00013+1

STO Temp+1

TSX	SUPLBN,4	Initialise standard locations for
PZE	ba+offsa	holding lower bounds and upper
OCT	020400000002	bounds, with those of the first
		array operand.
TSX	CHKBND,4	
TRA	L01002	
PZE	3	
PZE	bc+offsc	Check other arrays (taking part as
OCT	010203040502	operands) for having lower and upper
PZE	ba+offsa	bounds equal to those of the first
OCT	010300000002	array operand.
PZE	bb+offsb	
OCT	010300000002	
L01002	EQU *	
	USE JUNK	
L01003	EQU *	Subscript list for recursive address
PZE	bi+offsi	calculation of A(I,*,J,*).
PZE	bj+offsj	
L01004	PZE L00101	List of addresses where starting
PZE	0I1010	address and increments will be
PZE	0I2010	filled in by RLRCAC routine.
	USE CODE	
TSX	RLRCAC,4	Output for calling of recursive
PZE	L01003	address (real) calculation initiali-
PZE	ba+offsa,,L01004	sation routine for A(I,*,J,*).
OCT	020400000002	
	USE JUNK	
L01005	EQU *	
L01006	PZE L00102	
	PZE 1I1010	
	PZE 1I2010	
	USE CODE	
TSX	RLRCAC,4	Calling of RLRCAC routine for
PZE	L01005	initialisation of starting address
PZE	bc+offsc,,L01006	and increments for the recursive
OCT	010203040502	address calculation of C.
	USE JUNK	
L01007	EQU *	
PZE	bk ₁ +offsk ₁	
PZE	bk ₂ +offsk ₂	

LO1010 PZE L00103
 PZE 2I1010
 PZE 2I2010
 USE CODE
 TSX RLRCAC,4
 PZE L01007
 PZE ba+offsa,,L01010
 OCT 010300000002
 USE JUNK

For recursive address calculation
 of $A(*,K1,*,K2)$.

LO1011 EQU *
 PZE bi+offsi
 PZE bj+offsj

LO1012 PZE L00104
 PZE 3I1010
 PZE 3I2010
 USE CODE
 TSX RLRCAC,4
 PZE L01011
 PZE ba+offsa,,L01012
 OCT 020400000002
 USE JUNK

For recursive address calculation
 of $A(I,*,J,*)$.

LO1013 EQU *
 PZE bk₂+offsk₂

LO1014 PZE L00105
 PZE 4I1010
 PZE 4I2010
 USE CODE

TSX RLRCAC,4
 PZE L01013
 PZE bb+offsb,,L01014
 OCT 010300000002
 USE JUNK

For recursive address calculation
 of $B(*,K2,*)$.

LO1015 EQU *

LO1016 PZE L00106
 PZE 5I1010
 PZE 5I2010
 USE CODE

TSX RLRCAC,4
 PZE L01015
 PZE bc+offsc,,L01016
 OCT 010203040502

For recursive address
 calculation of $C(*,*)$.

	TRA	STC010	Transfer straight to the starting of repetitive codes.
1L0010	CLA	LB1.1	LB1.1 contains lower bound (first copy) for the first '*' in array cross-section reference.
	ADD	=1	
	STO	LB.1	
	SUB	UB.1	UB.1 contains the upper bound for the first '*' in array cross-section.
	TZE	100010	Transfer out of this particular loop.
0I1010	LXA	L00101,2	Get the next element address for
	TXI	*+1,2,**	A(I,*,J,*).
	SXA	L00101,2	
1I1010	LXA	L00102,2	For C.
	TXI	*+1,2,**	
	SXA	L00102,2	
2I1010	LXA	L00103,2	For A(*,K1,*,K2).
	TXI	*+1,2,**	
	SXA	L00103,2	
3I1010	LXA	L00104,2	For A(I,*,J,*)
	TXI	*+1,2,**	
	SXA	L00104,2	
4I1010	LXA	L00105,2	For B(*,K2,*)
	TXI	*+1,2,**	
	SXA	L00105,2	
5I1010	LXA	L00106,2	For C(*,*)
	TXI	*+1,2,**	
	SXA	L00106,2	
	TRA	STC010	
2L0010	CLA	LB1.2	LB1.2 contains lower bound (first copy) of the second '*' position in array cross-section reference.
	ADD	=1	
	STO	LB1.2	
	SUB	UB.2	UB.2 contains corresponding upper bound.
	TZE	200010	
0I2010	LXA	L0101,2	Getting address of the next element of A(I,*,J,*).
	TXI	*+1,2,**	
	SXA	L00101,2	

1I2010	LXA	LOO102,2	
	TXI	*+1,2,**	For C.
	SXA	LOO102,2	
2I2010	LXA	LOO103,2	
	TXI	*+1,2,**	For A(*,K1,*,K2).
	SXA	LOO103,2	
3I2010	LXA	LOO104,2	
	TXI	*+1,2,**	For A(I,*,J,*).
	SXA	LOO104,2	
4I2010	LXA	LOO105,2	
	TXI	*+1,2,**	For B(*,K2,*).
	SXA	LOO105,2	
5I2010	LXA	LOO106,2	
	TXI	*+1,2,**	For C(*,*).
	SXA	LOO106,2	
STC010	EQU	*	Starting of repetitive codes.
LOO105	CLA	**	
	FAD	Temp+0	
	STO	Temp+0	
	CLA	Temp+1	
	STO	Temp+1	Adding of B(*,K2,*) with the partial result.
LOO103	CLA	Temp+0	
	FAD	**	
	STO	Temp+0	
	LAC	LOO103,2	
	CLA	Temp+1	Adding of A(*,K1,*,K2) with the partial result.
	FAD	1,2	
	STO	Temp+1	
LOO104	CLA	Temp+0	
	FAD	**	
	STO	Temp+0	
	LAC	LOO104,2	
	CLA	Temp+1	Adding of A(I,*,J,*) with the partial result.
	FAD	1,2	
	STO	Temp+1	
LOO102	CLA	Temp+0	
	STO	**	
	LAC	LOO102,2	
	CLA	Temp+1	Assigning the result of the expression to C.
	STO	1,2	
LOO101	CLA	Temp+0	
	STO	**	
	LAC	LOO101,2	
	CLA	Temp+1	Assigning the result of the expression to A(I,*,J,*).
	STO	1,2	

TRA 2L0010	Repeat the loop for next array element in the cross-sections.
200010 CLA LB2.2	LB2.2 contains the second copy of lower bound for the second '*' in array cross-section reference.
STO LB1.1	
TRA 1L0010	Repeat the loop for next cross-section (i.e. for the next subscript value of the first '*').
100010 EQU *	

Note: Even though for bound checking only one of the two references of A(I,*,J,*) was supplied, for recursive address calculation the two references were treated to be different.

APPENDIX M

M-1: Coding of an Edit Directed Output Statement with Nested 'DO' Groups:

The statement:

```

PUT (FILE1) SKIP (<exp1>) EDIT(((A,B,<exp2> DO
                                2 1
I=1 TO M BY N WHILE C>0), P, { (Q,R,S DO J=N TO L1 BY L2
                                1' 4 3
WHILE CDASH<0) DO K=MM TO NK BY KK WHILE K2<0) ,T, W
                                3' 4'
DO L(3) = J,K, 1 TO 10 BY 2) ) (<immediate format item>);
                                2'

```

In the above statements the first order storage addresses are as given below:

A ... BS.01+000001	L1 ... BS.01+000012
B ... BS.01+000002	L2 ... BS.01+000013
C ... BS.01+000003	CDASH BS.01+000014
M ... BS.01+000004	K ... BS.01+000015
N ... BS.04+000005	KK ... BS.01+000016
P ... BS.01+000006	T ... BS.01+000017
Q ... BS.01+000007	W ... BS.01+000018
R ... BS.01+000008	MM ... ES.01+000019
S ... BS.01+000009	NK ... BS.01+000020
I ... BS.01+000010	K2 ... BS.01+000021
J ... BS.01+000011	

We assume that B,P,Q and R are formal parameters and the remaining an ordinary variables. Further W is an array of dimension 3. Those variables starting with I,J,K,L,M and N are integers while the rest are floating point numbers.

<u>The Map Code</u>			<u>Comments</u>
	TSX	.IOSUP,4	Open file and validate file operation.
	PON	PLF.OO,,F.OO	
	EXTERN	.IOSUP	
arithmetic code for exp1			
	TSX	IOHSC.,4	Execute SKIP command.
LO0001	PZE	**	
	EXTERN	IOHSC.	
	TRA	MO0001	G · for performing format linkage.
MO0002	EQU	*	
	TRA	MO0004	Transfer to coding of 'DO' Group 2-2'
MO0006	TRA	**	DOMAIN of 'DO' Group 1-1' begins.
	AXT	BS,01+000001,1	
	PXA	,1	
	TSL	HNLIO.	Output variable A.
	EXTERN	HNLIO.	
	CLA	BS.04+000002	
	TSL	HNLIO.	Output variable B.
arithmetic code for exp2			
LO0002	AXT	**,1	
	PXA	,1	
	TSL	HNLIO.	Output expression.
	TRA	MO0006	DOMAIN OF 'DO',Group 1-1' ends.
MO0005	TRA	**	DOMAIN of 'DO' Group 2-2' begins.
	CLA	=1	Coding of 'DO' Group 1-1' begins.
	STO	BS.01+000010	Initialize 'DO' index.
	CLA	BS.01+000004	

STO	TS.01+000001	Save upper limit in temporary.
CLA	BS.01+000005	
STO	TS.01+000002	Save increment in temporary
X00001 CLA	TS.01+000001	check if 'DO' index has crossed
SUB	BS.01+000010	the limit.
TMI	Y00001	

arithmetic coding for while
clause. Requires two labels.

TXXXXX and FXXXXX

TXXXXX TSL	M00006	Call to Domain of 'DO' Group 1-1'
CLA	BS.01+000010	increment 'DO' index.
ADD	TS.01+000002	
STO	BS.01+000010	
TRA	X00001	
FXXXXX EQU	*	
Y00005 EQU	*	Coding of 'DO' Group 1-1' ends.
CLA	BS.01+000006	
TSL	HNLIO.	Output variable P.
TRA	M00007	Transfer to coding of 'DO' Group 4-4'
M00011 TRA	**	Domain of 'DO' Group 3-3' begins.
CLA	BS.01+000007	
TSL	HNLIO.	Output variable Q.
CLA	BS.01+000008	
TSL	HNLIO.	Output variable R.
AXT	BS.01+000007,1	
PXA	,1	
TSL	HNLIO.	Output variable S.
TRA	M00011	Domain of 'DO' Group 3-3' ends.
M00010 TRA	**	Domain of 'DO' Group 4-4' begins.
CLA	BS.01+000005	Coding of 'DO' Group 3-3' begins
STO	BS.01+000011	initialize 'DO' index.
CLA	BS.01+000012	
STO	TS.01+000001	Save upper limit.
CLA	BS.01+000013	
STO	TS.01+000002	Save increment.

X00002 CLA TS.01+000001 Check if 'DO' index has crossed
 SUB BS.01+00011 the upper limit.

TMI Y00002

arithmetic code for while
 clause. Requires the
 labels

TYYYYY and F YYYYY

TYYYYY TSL M00011 Call domain of 'DO' Group

CLA BS.01+000011

ADD TS.01+000002 Increment index.

STO BS.01+000011

TRA X00002

FYYYYY EQU *

Y00002 EQU *

TRA M00010 Coding of 'DO' group 3-3' and
 Domain of 'DO' group 4-4' end.

M00007 EQU *

CLA BS.01+000019

STO BS.01+000015 Initialize 'DO' index.

CLA BS.01+000020

STO TS.01+000001 Save upper limit

CLA BS.01+000016

STO TS.01+000002 Save increment

X00003 CLA TS.01+000001 Check if 'DO' index has
 crossed up per limit.

SUB BS.01+000015

TMI Y00003

arithmetic coding of while
 clause; requires two labels

TZZZZZ and FZZZZZ

TZZZZZ TSL M00010 Call domain of 'DO' Group 4-4'.

CLA BS.01+000015

ADD TS.01+000002 Increment index.

STO BS.01+000015

TRA X00003

FZZZZZ EQU

Y00003 EQU	*	Coding of 'DO' Group 4-4' ends.
AXT	BS.01+000017,1	
PXA	,1	
TSL	HNLIO.	Output variable T.
TSX	F.RNGE,4	To calculate the starting address,
PZE	BS.01+00018,,	increment and range for array W.
ETC	Z00004	
PZE	Z00005,,Z00006	
Z00004 AXT	**,1	
PXA	,1	
TSL	HNLIO.	
Z00005 TXI	*+1,1,**	
SXA	Z00004,1	
Z00006 TXL	Z00004,1,**	
TRA	M00005	Domain of 'DO' Group 2-2' ends.
M00004 EQU	*	Coding of 'DO' Group 2-2' begins.

arithmetic coding for
calculating address
of L(3)

I00003 AXT	**,1	
SXA	TS.01+000001,1	Store address of 'DO' index.
CLA	BS.01+000011	
STO*	TS.01+000001	Initialize 'DO' index.
TSL	M00005	Call Domain of 'DO' Group 2-2'.
CLA	BS.01+000015	
STO*	TS.01+000001	Initialize 'DO' index.
TSL	M000005	Call Domain of 'DO' Group 2-2'.
CLA	=1	
STO*	TS.01+000001	Initialize 'DO' index.
CLA	=10	
STO	TS.01+000002	Save upper limit.
CLA	=2	
STO	TS.01+000003	Save increment.

X00007	CLA	TS,01+000002	Check if 'DO' index has
	SUB*	TS,01+000001	crossed the limit.
	TMI	Y00007	
	TSL	M00005	Call domain of 'DO' Group 2-2'.
	CLA*	TS,01+000001	
	ADD	TS,01+000003	
	STO*	TS,01+000001	
	TRA	X00007	
X00007	EQU	*	
	TRA	M00003	Get out of output statement.
M00001	TSX	STHIO.,4	call format link age routine.
	PZE	M00012	
	TRA	M00002	Go to beginning of Edit Statement
M00012	EQU	*	

format list

M00003 EQU

:
:
:

M-2: Coding of a DO statement outside an I/O command:

The statement:

DO I = 1,2,3, 10 TO 20 BY 2, M TO N BY K
WHILE B<0;

First order address of I is BS.01+000001, of M is
BS.01+000002, of N is BS.01+000003, of K is BS.01+000004.
Let the DO serial number be 2.

CLA	=2	Update do-serial number.
STO	DOSRNO	Update DO-serial number.
CLA	=1	
STO	BS.01+000001	Initialize 'DO' index
TSL	C00002	Call domain of DO.

	CLA	=10	
	STO	BS.01+000001	Initialize 'DO' index.
	CLA	=20	
	STO	TS.01+000001	Save upper limit in temporary.
	CLA	=2	
	STO	TS.01+000002	Save increment in temporary
X00001	CLA	TS.01+000001	check if 'DO' index exceeds the
	SUB	BS.01+000001	upper limit.
	TMI	Y00001	
	TSL	C00002	Call domain of 'DO' Group.
	CLA	BS.01+000001	
	ADD	TS.01+000002	Increment 'DO' index and loop back.
	STO	BS.01+000001	
	TRA	X00001	
Y00001	EQU	*	
	CLA	BS.01+000002	Initialize 'DO' index.
	STO	BS.01+000001	
	CLA	BS.01+000003	
	STO	TS.01+000001	Save upper limit in temporary.
	CLA	BS.01+000004	
	STO	TS.01+000002	Save increment in temporary.
X00002	CLA	TS.01+000001	Check if 'DO' index has exceeded
	SUB	BS.01+000001	limit.
	TMI	Y00002	

while clause coding,
requires labels

TXXXXX and FXXXXX

TXXXXX	TSL	C00002	
	CLA	BS.01+000001	Increment 'DO' index, and loop.
	ADD	TS.01+000002	
	STO	BS.01+000001	
	TRA	X00002	
FXXXXX	EQU	*	
Y00002	EQU	*	
	TRA	P00002	Getout of 'DO' group.

000002 TRA *

Body of the 'DO' Group

TRA 000002

These two instructions are generated when 'end' of 'DO' Group is recognised.

P00002 EQU *

·
·
·

M-3: Input and Output Formats for Stream Oriented Data

Transmission in PL 7044:

List Directed Input:

Data in the input stream has one of the following general forms:

- a) +|- arithmetic constant.
- b) Character string constant.
- c) Bit string constant.
- d) +|- real constant +|- imaginary constant.

An arithmetic constant may be fixed point (integer) or floating point constant and may have binary or decimal representation.

Data in the data stream will be called source and the variable to which the data is to be assigned will be called target.

Data items in the stream must be separated by a blank or by a comma. The separator may be preceded and/or followed by an arbitrary number of blanks. A null field in the stream is indicated either by the very first non-blank character in the stream being a comma or by two adjacent commas separated by an

arbitrary number of blanks. The transmission of the list directed input is terminated by expiration of the data list or the end-of-data or end-of-file condition. In the latter case, the processing of the job will also be terminated.

If the data is a character string constant, the surrounding quotation marks are deleted and double quotation marks within the string are treated as single quotation marks. Character strings may not have a length greater than 128. Bit strings may not have a length greater than 36.

If data is an arithmetic constant or a complex constant, it is converted to coded arithmetic form. The target attributes are then examined and source conversion if necessary, is performed. It is then assigned to the target. The accompanying table gives the possible conversions.

List Directed Output:

The value of the scalar variable is converted to character representation and transmitted to the data stream. Two blanks always precede the transmission of a data value. If the length of a numeric data item is longer than the number of character positions remaining in the current line (record), the entire item is outputted starting at the beginning of the next record. Character strings whose length exceeds the size of a record will be split between two or more records. No blanks will be inserted when such an item is continued on the next line.

All fixed point arithmetic data items are outputted in I(11) format. All floating point arithmetic data items are printed in E(14,8) format. All complex data items are printed in 2E(14,8) format. For character and bit strings, the format is set according to their present length.

Data-Directed Input:

The data directed data in the input stream has the following format:

Scalar-variable = Constant \forall [, scalar variable =
 Constant] \$

Scalar variable may be subscripted name with optionally signed decimal integer constants for subscripts.

Each assignment may be separated by an arbitrary number of blanks or a comma.

The constant has the same format as in the case of list-directed input.

The scalar variable names used on the left hand side of the '=' sign must be one of the names used in the data list used in the corresponding GET DATA statement. The order in which names appear in the data-list need not be same as the order in which they appear in the stream nor is it necessary for all the names in the data list to appear in the data stream.

The data list may not include major or minor structures. Structure elements may be used, however, provided the number of characters forming the base element including the dot between qualifying names, does not exceed thirty.

Data Directed Output:

The general format for data directed output is the following:

Scalar variable = Constant $\delta\delta$ Scalar variable
 $\delta_1 = \delta$ constant ...

If an array occurs in the data list as a data element, then the elements with their subscripts are transmitted in row major order.

Each assignment is separated by two blanks. The constant has the same format as in the case of list-directed output.

Structures and 'DO' repetitive groups are not allowed.

The length of the data field in both input and output is a function of the length of the identifier and that of the associated data constant. If, during output transmission, the length of the data field exceeds the number of remaining character positions in the record (line) then a new line is started and the data item is then outputted on this line.

Edit Directed Input and Output:

The edit directed I/O format is governed by the associated format specifications whose syntax and semantics are given in the chapter on 'Analysis of Format Statements'.

TABLE OF CONVERSION FOR LIST DIRECTED INPUT

<div> <div>TARGET</div> <div>SOURCE</div> </div>	INTEGER	FL. POINT	COMPLEX	BIT ST.	CHAR ST.	BINARY INTEGER	BINARY FL. POINT
	INTEGER	FLOATING POINT	COMPLEX	BIT ST.	CHAR ST.	BINARY INTEGER	BINARY FL. POINT
INTEGER	✓	✓	✓ REAL PART ONLY	✓	✓ ONLY NUMERICS	✓	✓
FLOATING POINT	✓	✓	✓ REAL PART ONLY	✓	✓ ONLY NUMERICS 8, 11, E, I	✓	✓
COMPLEX	✓	✓	✓	✓	✓ ONLY NUMERICS 8, 11, E, I	✓	✓
BIT STRING	✓	X	X	✓	✓ ONLY 0'S & 1'S	✓	X
CHAR STRING	✓	✓	✓	✓	✓	✓	✓

AP.

✓ : CONVERSION POSSIBLE X : CONVERSION NOT POSSIBLE

APPENDIX N

Output of Format Generator:

Input:

(5((M)(X(10), COLUMN(M), C(F(10,M),E(M,<exp>)),
(M)I(M), 'RESULT', A(<exp>), B(3),5Ø(12))))).

The MAP code generated will be as follows:

AXT	5,1
TSL	IOHLP.
EXTERN	IOHLP.

CIA*	BS.02+000100
AXT	0,1
TMI	*+2
PAX	,1
TSL	IOHLP.

TSX	IOHXC.,4
PZE	10
EXTERN	IOHXC.

CIA*	BS.02+000100
TPL	*+2
ZAC	
STA	*+2
TSX	IOHCC.,4
PZE	**
EXTERN	IOHCC.

CIA*	BS.02+000100
AXT	0,4
TMI	*+2
PXA	,4
SXD	*+2,4
TSX	IOHFC.,4
PZE	10,,**
EXTERN	IOHFC.

[arithmetic coding]

CIA*	BS.02+000100
AXT	0,1
TMI	*+2
PAX	,1
SXA	*+2,1
TSX	IOHEC.
PZE	
EXTERN	IOHEC.

LOOOO1

CLA*	BS.02+000100
AXT	0,2
TMI	*+2
PAX	,2

CLA*	BS.02+000100
TPL	*+2
ZAC	
STA	*+2
TSX	IOHIC.,4
PZE	**
EXTERN	IOHIC.,4
PZE	**
EXTERN	IOHIC.
PZE	
TSX	IOHHC.,4
PZE	6
BOI	1,RESULT
EXTERN	IOHHC.

[arithmetic coding]

	TSX	IOHAC.,4
100002	PZE	
	EXTERN	IOHAC.

	TSX	IOHBC.,4
	PZE	3
	EXTERN	IOHBC.

	AXT	5,2
--	-----	-----

	TSX	IOHOC.,4
	PZE	12
	EXTERN	IOHOC.

	TSL	IOHRP.
	EXTERN	IOHRP.

	TSL	IOHRP.
--	-----	--------

	TRA	IOHEF.
--	-----	--------

APPENDIX Ø

Run Time Stack Management Routines (RTSMR):

a. List of RTSMR:

- | | |
|---|---------|
| 1. Open a begin block | R.OPBG |
| 2. Invoke a procedure block | R.OPPR |
| 3. Terminate a block by 'END'
Statement | CLSEND |
| 4. Terminate a block by RETURN
statement. | CLPRWT |
| 5. Terminate a block by RETURN
statement and return a value. | CLPRRT |
| 6. GOTO statement with block
closing. | GTBLOK. |

b. Link Cell Format:

	L.P.P.		ret addr
	W.A.P.		≠ Saved
	DOSRNO		Calling block no.

where,

P.P.P. (Present Procedure Pointer): This is a one word area which contains the address of the first of the 3 locations which constitute the linkage cell of the block which is being currently executed.

L.P.P. (Last Procedure Pointer): This is the value of the P.P.P. when the invocation of the present block was made.

W.A.P. (Work Area Pointer): It is the address of the first free location after arrays and character variables (Second order storage area) have been allocated memory space. This denotes the address of the first free cell available for assigning temporaries requiring more than 2 words.

F.C.P. (Free Cell Pointer): It is the address of the first free location after multiword temporaries have been assigned.

~~/~~ Saved: It is the number of first order locations saved, if any, when the present block was invoked.

ret addr (return address): It is the address of the calling point, if the invoked block is a PROCEDURE block. It is zero for a BEGIN block. The return pointer is needed at the time of terminating the block.

Calling block no.: It is the serial number of the calling block.

DOSRNO: It is the serial number of the innermost DO-group open at the calling point, if any.

PSRNO.: Global location which contains the serial number of the innermost block open.

Notation:

(LNKCEL, i) i=0,1,2 represents the (i+1)-st word of
the linkage cell.

A is used to represent the address portion.

D is used to represent the decrement portion.

LNKCEL is same as F.C.P.

1. R.OPBG:

Calling sequence: AXT blkopn,1
 TSX R.OPBG,4

where,

'blkopn' is the block number of the block to be opened.

Actions Taken:

BLKOPN	←XR1	index register '1'
D(LNKCEL,0)	←P.P.P.	
A(LNKCEL,0)	←0	
P.P.P.	←F.C.P.	
F.C.P.	←F.C.P.+3	
D(LNKCEL,1)	←W.A.P.	
A(LNKCEL,1)	←0	
D(LNKCEL,2)	←DOSRNO	
A(LNKCEL,2)	←PSRNO.	
PSRNO.	←BLKOPN	

(Some of the pointers are being saved unnecessarily
because it shares the routine with R.OPPR.)

Return of Control:

TRA 1,4

2. R.OPPR:Calling Sequence:

```

TSX  R.OPPR,4
TXI  *+3,,arglst
PZE  PHWadd
PZE  tempad,, mnrtyp

```

for procedure invocation by a CALL statement the last word is

```
PZE  0
```

where,

'arglst' is the address of the argument list, if any.

'PHWadd' is the address of the procedure header word.

'tempad' is the address of the result receiving temporary

'mnrtyp' is the minor type of the result returned.

Actions Taken:

1. Extract the number of the block to be opened from the PHW and, store it in BLKOPN.

2. Determine if any saving is to be done.

$BS(\text{blkopn}) - BE(\text{psrno}) \geq 0 \Rightarrow \text{no saving}$

$< 0 \Rightarrow \text{saving is to be done}$

If saving is done, update S.A.P. accordingly.

$\# \text{ saved} = |BS(\text{blkopn}) - BE(\text{psrno})|$

$S.A.P. \leftarrow S.A.P. + \# \text{ Saved.}$

3. $D(LNKCEL, 0) \leftarrow P.P.P.$

$A(LNKCEL, 0) \leftarrow XR4$

Contents of Index

Register 4.

W.A.P. $\leftarrow D(LNKCEL,1)$

PSRNO. $\leftarrow A(LNKCEL,2)$

DOSRNO $\leftarrow D(LNKCEL,2)$

Return:

TRA 1,4

2.2 For Terminating a PROCEDURE block:

(Entry point CLEND.)

- a. $A(LNKCEL,1) = 0, \Rightarrow$ no restoring to be done
 $\neq 0 \Rightarrow$ restoring first order storage area
to be done.

Start address of the area to be restored is $BS.(PSRNO.)$

It is determined from the storage table. The restoration starts
from S.A.P. and goes to S.A.P. - \neq saved.

At the end of restoration, update S.A.P. as

S.A.P. \leftarrow S.A.P. - \neq saved

- b. $DOSRNO \leftarrow D(LNKCEL,2)$

If $DOSRNC \neq 0$, the DO-groups that were closed at the
time of invoking this procedure are opened once again i.e. their
do-status bits are put ON.

XR 4 $\leftarrow A(LNKCEL,0)$

- c. If (3,4) = 0, no result is expected
 $\neq 0$, result is expected, hence error.


```

d.          F.O.P.          ← P.P.P.
            P.P.P.          ← D(LNKCEL,0)
            W.A.P.          ← D(LNKCEL,1)
            PSRNO.          ← A(LNKCEL,2)

```

Return:

TRA 1,4

4. CLPRWR:

Calling sequence:

TSX CLPRWR, 4

Actions taken:

LOOP CIA* P.P.P.

```
TMI      Error      Trying to close the main
                        procedure by a RETURN statement
```

PAX , 4

TXH CLEND., 4, 0

AXT 1-LOOP,4

TRA CLEND 1

CLEND. and CLEND1 are two entry points in the CLEND routine, which skip some of the unwanted code. It can be seen from the present scheme, that all the BEGIN blocks, those are open, also get closed. This process is repeated till a procedure block invocation is obtained.

5. CLP RRT:

Calling sequence

TSX CLPRRT,4

PZE tempadd,,mnrtyp

a. First it determines the result receiving temporary, if any, and the minor type of result expected. Now it finds out whether any conversion is to be done. After doing the necessary conversion, value returned is assigned to result temporary.

b. Now it merges with CLPRWR with the difference that error clause in 3.b gets nullified.

6. GTBLOK:

Calling sequence

```
TSX      GTBLOK,4
PZE      label
PZE      sblno,, dblno
```

where,

'sblno' is the source block number.

'dblno' is the destination block number.

Actions:

Close block by calling CLSEND routine at the CLEND entry point till the destination block number is obtained.

Return:

TRA* 1,4

APPENDIX P

FREE LIST:

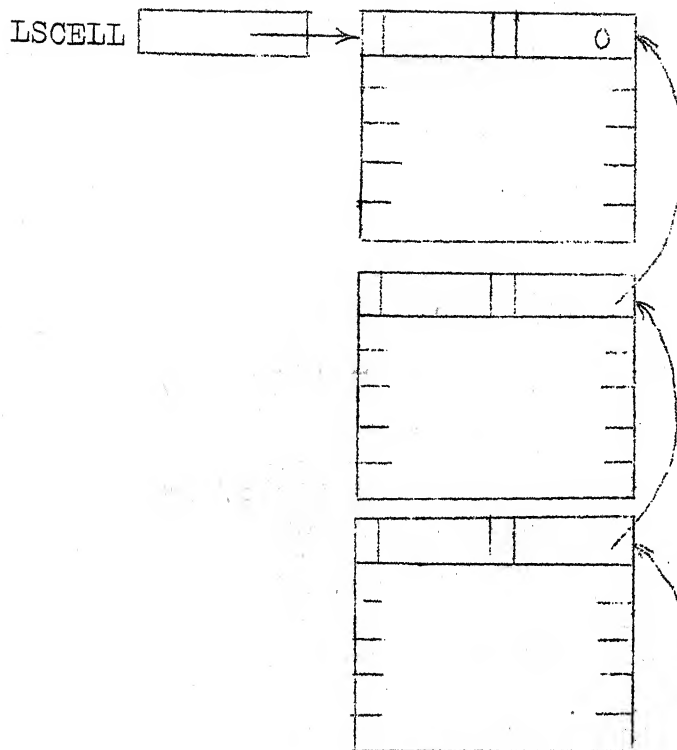
Free core area, in second pass, has been divided into six word cells. These cells are needed for multiword temporary storage by a number of routines viz. expression processor, 'BY NAME' option analysis routine, symbol table etc. To realize economy in storage space, a linked list is maintained from which cells can be taken when needed and to which cells could be returned when no more needed. This ensures that no cell is lost;

Since symbol table makes maximum use of the free list, the size of a symbol table cell fixed the size of the attribute cell.

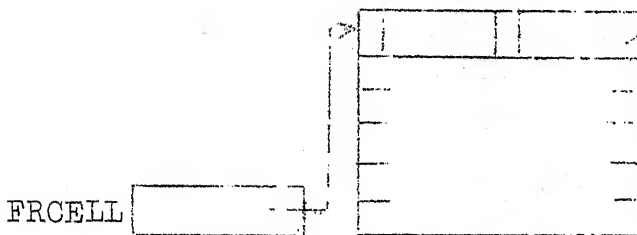
The stack-discipline of the free list is last in first out (LIFO). A cell which is returned to the free list is added to one end of the free list. The cell which was last added would be the first to be given on a call for a new cell.

Two pointers (FRCCELL, LSCCELL) are maintained which store the address of the cells which is to go out first/to which the cell being returned would be chained and which is at the other end of the chain respectively. In case the latter cell is taken, list becomes empty.

The format of the free list is shown in Fig. P-1:



Last cell.
List becomes empty,
if it is taken.



First cell to be given
to the calling routine
(if no cell is added in
between).

Fig. P-1: Free List.

APPENDIX Q

PL 7044 Compiler Output for a PL/I Program:

Let us consider the following PL/I program for computing the factorial of an integer.

```

MAIN: PROCEDURE OPTIONS (MAIN);
      /* RECURSIVE COMPUTATION OF FACTORIAL */
      GET LIST (M); /* READ THE VALUE OF M */
      PUT EDIT ((N, FACT(N) DO N = 1 TO M))
              (COLUMN (1) 'FACTORIAL OF 'I(4) '='
              E(14,8));

FACT: PROCEDURE (X) RETURNS (REAL (10,1)) RECURSIVE;
      DECLARE X FIXED (8);
      IF X = 1 THEN RETURN(1.);
      /* DUMMY ELSE TO BE GENERATED */
      RETURN (X* FACT (X-1));
      END MAIN; /* MULTIPLE CLOSURE */

```

The MAP code generated for this program is as follows:

Following code is generated in the second pass

\$IBMAP PL7044		
USE	BSS	
USE	JUNK	
USE	PROLOG	
USE	NAME	
USE	STATIC	Common to every PL/I program.
USE	CODE	
EXTERN	R.INIT	
USE	STATIC	
START TSL	R.INIT	
L00000 EQU	=1.0	
L00001 EQU	=1B1	
L00002 EQU	=1	
USE	JUNK	
L00005 VFD	03/1,15/000000,03/0,15/L00003	PHW for
L00011 VFD	03/2,15/L00010,03/0,15/L00007	PHW for
USE	PROLOG	
L00010 EQU	*	Prolog list for FACT
VFD	3/0,15/BS.02+000000,6/1,6/7,6/5	
PZE	0	End of prologue list
L00013 EQU	=02014000000000	Literal 1.0
L00014 EQU	=02014000000000	Literal 1.0
L00015 EQU	=00000000000001	Literal 1

Following code is generated in the third pass.

```

LO0003 EQU      *           Entry point for MAIN
      TSL      DCL.01
      TRA      DCE.01
DCL.01 PZE      **          Closed procedure for declaration
      TRA*     DCL.01       of block MAIN.
DCE.01 EQU      *
LO0006 EQU      *
      USE      JUNK
S.FSBN PON      0           File status block for system
      BSS      6           input file
      PZE      0
      USE      CODE
      TSX      .IOSUP,4     Call to I/O supervisor
      PZE      S.FBIN,,S.FSBN
      EXTERN   .IOSUP
      TSX      G.LIST,4     Call to list directed inputting
                           routine.
      PZE      BS.01+000000 'address of M.
      EXTERN   G.LIST
      USE      JUNK
S.FSBU PZE      0           File status block for system
      BSS      6           output
      PZE      0
      USE      CODE
      TSX      .IOSUP,4     Call to I/O Supervisor
      PON      S.FBOU,,S.FSBU
      TRA      MO0000       Go and perform format linkage
MO0001 EQU      *
      TRA      MO0003       Goto coding of 'DO'.
MO0004 TRA      **          Domain of 'DO' starts.
      AXT      BS.01+000001,1
      PXA      ,1
      TSL      HNLIO.       Output N
      EXTERN   HNLIO.
      USE      JUNK
LO0017 EQU      *           Argument list for function call FACT
      PZE      BS.01+000001
      USE      CODE
      EXTERN   R.OPPR
      TSX      R.OPPR,4     Call to function FACT
      TXI      *+3,, LO0017
      PZE      LO0011
      PZE      TS.01+000000,,2
      AXT      TS.01+000000,1
      PXA      ,1
      TSL      HNLIO.       Output FACT(N)
      TRA      MO0004       Eng of Domain of 'DO'

```

M00003	EQU	*	Coding of 'DO'.
	CIA	=1	
	STO	BS.01+000001	Initialise index
	CIA	BS.01+000000	
	STO	TS.01+000001	Save upper limit.
	CIA	=1	
	STO	TS.01+000002	Save increment.
X00000	CIA	TS.01+000001	Check if index has
	SUB	BS.01+000001	crossed limit.
	TMI	Y00000	Go out of 'DO' loop
	TSL	M00004	Transfer to Domain
	CIA	BS.01+000001	
	ADD	TS.01+000002	
	STO	BS.01+000001	
	TRA	X00000	
Y00000	EQU	*	
	TRA	M00002	
M00000	EQU	*	
	TSX	TSHIO.,4	Format list starts.
	PZE	M00005	
	TRA	M00001	
	EXTERN	TSHIO.	
M00005	EQU	*	
	TSX	IOHCC.,4	
	PZE	1	
	EXTERN	IOHCC.	
	TSX	IOHHC.,4	
	PZE	13	
	BCI	3, FACTORIAL OF	
	EXTERN	IOHHC.	
	TSX	IOHIC.,4	
	PZE	4	
	EXTERN	IOHIC.	
	TSX	IOHHC.,4	
	PZE	1	
	BCI	1,=	
	TSX	IOHEC.,4	
	PZE	14,8	
	EXTERN	IOHEC.	
	TRA	IOHEF.	Format list ends.
	EXTERN	IOHEF.	
M00002	EQU	*	
	TRA	END.02	Tra around procedure FACT
L00007	EQU	*	Entry point for FACT.
	TSL	DCL.02	
	TRA	DCE.02	
DCL.02	PZE	**	Closed procedure for
	TRA*	DCL.02	declarations of FACT.
DCE.02	EQU	*	
L00012	EQU	*	
	CIA*	BS.02+000000	Coding for IF statement
			starts.

INTFLT	MACRO		Macro for integer
	ORA	=02330000000000	to floating point
	FAD	=02330000000000	conversion
	ENDM		
	INTFLA		
	CAS	L00013	Literal 1.0
	TRA	F00016	
	TRA	S00016	
	TRA	F00016	
S00016	EQU	S.0001	
F00016	EQU	F.0001	
S.0001	EQU	*	
	EXTERN	CLPRRT	
	TSX	CLPRRT,4	Code for RETURN (1.)
	PZE	L00014,,2	Literal 1.
	TRA	E.0001	
F.0001	EQU	*	
E.0001	EQU	*	
	CLA*	BS.02+000000	
	SUB	L00015	Literal 1
	STO	TS.02+000000	
	USE	JUNK	
L00020	EQU	*	Argument list for function call FACT (X-1).
	PZE	TS.02+000000	
	USE	CODE	
	TSX	R.0PPR,4	
	TXI	*+3,,L00020	Call to FACT
	PZE	L00011	
	PZE	TS.02+000000,,2	
	CLA*	BS.02+000000	
	INTFLT		
	STO	TS.02+000001	
	LDQ	TS.02+000001	
	FMP	TS.02+000000	
	STO	TS.02+000000	
	TSX	CLPRRT,4	Code for RETURN(X*FACT(X-
	PZE	TS.02+000000,,2	
	EXTERN	CLSEND	
	TSX	CLSEND,4	END for FACT
END.02	EQU	*	
	TSX	CLSEND,4	END for MAIN
END.01	EQU	*	

Following code is generated at the end of third pass.

	ENTRY	BLPDTB	
BLPDTB	EQU	*	Block predecessor table
	OCT	00000000000000	
	OCT	00000000000000	
	OCT	00000000000001	
	ENTRY	STGTAB	Block storage table
STGTAB	EQU	*	
	PZE	BE.00,,BS.00	
	PZE	BE.01,,BS.01	
	PZE	BE.02,,BS.02	
AS.00	EQU	BS.00+000000	
TS.00	EQU	AS.00+000000	
BE.00	EQU	TS.00+000000	
BS.01	EQU	BE.00	
AS.01	EQU	BS.01+000002	
TS.01	EQU	AS.01+000000	
BE.01	EQU	TS.01+000003	
BS.02	EQU	BE.01	
AS.02	EQU	BS.02+000001	
TS.02	EQU	AS.02+000000	
BE.02	EQU	TS.02+000002	
BS.00	EQU	*	
	BSS	000008	
	ENTRY	DOTBL	Do predecessor table start
DOTBL	EQU	*	
	END	START	

APPENDIX R

Glossary of Important Terms and Words:

Activating a 'BEGIN' block/procedure block:

A BEGIN block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when procedure is invoked at any one of its entry points.

Asynchronous Operations:

See Synchronous operations.

Attribute: An attribute is a descriptive property associated with a name to describe a characteristic of a data item or file .. which the name may represent.

Base Element: See structure.

Begin Block: See Block.

Block: A block is a collection of statements that defines the program region .. or scope throughout which an identifier is established as a name.

A BEGIN block can be activated only by the normal sequential flow of the program and can be used wherever a statement can be used.

A procedure block can only be activated remotely by CALL statements or by function references.

Body of a Block: The body of a block is defined as the collection of statements which appear lexicographically between the heading statement and the trailing statement of the block.

Compile Time: This is the period during which a program is being compiled.

Compile Time Data Table:

This is a fixed table of size 500 words used for storing data elements appearing in data directed I/O.

Data List: A data list is a list of data elements appearing in an I/O statement.

Data Set: A collection of data external to the program constitutes a data set.

Deactivating a Block:

A block is said to be deactivated when control is transferred to any one of the following statements (a) END (b) STOP (c) RETURN (valid only in case of a procedure), (d) GOTO statement whose destination is a point outside the block.

DO Index: A scalar variable which is used for controlling the number of times the domain the 'DO' is to be executed is known as the 'DO' index.

DO Parameter/Specification:

A 'DO' specification is defined as

expression -1 $\left[\begin{array}{l} \text{TO expression-2} \quad [\text{BY expression-3}] \\ \text{BY expression-3} \quad [\text{TO expression-2}] \end{array} \right]$

$[\text{WHILE (expression)}]$

Domain of 'DO': This is the collection of statements which appear lexicographically between the 'DO' statement and the corresponding END statement.

Dynamic Descendence:

If a block B is active, and another block B1 is activated from a point internal to block B (while B still remains active, then B1 is said to be the immediate dynamic descendent of B. If block B1 has a immediate dynamic descendant in B2, then B2 is said to be the dynamic descendent of B.

Dynamic Encompassing:

If block B is a dynamic descendent of block A, then block A dynamically encompasses block B and block B is dynamically encompassed. by A.

Epilogue: See prologue.

Expression: An expression is an algorithm used for computing a value.

Scalar Expression: A scalar expression is one that has a scalar value.

Aggregate Expression: An aggregate expression is an expression involving one or more aggregate operands i.e. structure or array.

Execution Time: This is the period during which the control is with a) the object code generated by the compiler and b) the supporting library routines.

First Order Storage Area:

This is the area where all scalar variables both STATIC and AUTOMATIC, array header word(s), dope vector words and single and double word temporaries.

Field Count: This is the number of times a format item is to be repeated.

Fixed Table: A binary search table of all the keywords (or pseudo reserved words), builtin procedure names and pseudo variable names.

Format Explicitor:

The entity defines the field width in case of the following: A-, B-, I-, X-, O-, COLUMN-, LINE-, and SKIP- format items. And in case of E- and F- format items, the first explicitor defines the field width and the second explicitor defines the number of places after the decimal point.

Group Count: It is the number of times a group of format items have to be repeated.

Implementation Dependency:

There are certain aspects of a language which cannot be rigorously defined without recourse to the characteristics of the machine on which the language is to be implemented. Such features are said to be implementation dependent.

Immediate Predecessor:

See predecessor

Immediate Dynamic Descendant:

See Dynamic Descendant.

Linear Space: This is a concept wherein the mapping between the memory space and the name space is the identity function. The addresses given by the loader in the name space are the same as in the memory space.

Lexical Unit: The program source text is broken up into units having unique codes recognisable by the syntax analysing routines. Such units are called lexical units.

Pass: A pass is defined as a scan through the text of the program. The text may be the source text or the text prepared by some other pass. During the scan, the text is modified and a new text is produced.

Partitioning of Memory:

In this report, partitioning of memory refers to software division of available core space. In PL 7044, the core space has been divided into the following: Nucleus and IOCS, First Order Storage, Second Order Storage, Third Order Storage, Object Code, Library Routines, and I/O Buffer Area.

Predecessor:

A is said to be the predecessor of B if B appears lexicographically between the heading statement and the trailing statement of A. If in addition, A is closest to B (lexicographically) then A is said to be the immediate predecessor of B.

Primary entry Point:

A primary entry point is that entry point identified by the left most entry on the PROCEDURE Statement. All other names for the procedure identify the secondary points.

Prologue: On entering a block certain initial actions are performed. These initial actions constitute the prologue. Actions that are performed when a block is deactivated constitute the epilogue.

Pseudo Variables:

In general, pseudo variable are built-in functions that can appear wherever other variables can appear in order to receive values. However, in PL 7044 the use of pseudo variables is much more restrictive.

Recursion: Recursion is said to have taken place when a procedure is reactivated while it is still active.

Run Time Stack: The second and third order storage share the free core available after loading the object program in what may be termed a double ended stack. This stack is referred to as the run time stack.

Secondary Entry Point:

See primary entry point.

Second Order Storage:

It is the storage space occupied by the elements of an array, character scalar variables, and multiword (more than 2 words) temporaries.

Structure: A structure is a hierarchical collection of scalar variables, arrays and structures. The outer most structure is called the major structure. All contained structures are called minor structure.

Base Element: Elements of structures which are themselves not structures are known as base elements.

Synchronous Operation:

A program in which the execution of statements proceeds serially in time is said to be executed synchronously.

When several statements are executed in parallel in time, then the operation is said to be asynchronous.

TASK:

A task is an identifiable execution of a set of instructions and exists only during the execution of the set of instructions. Note that TASK is not a set of instructions but the execution of the instructions.

BIBLIOGRAPHY

1. PL/I Language specification - IBM; Order Number GY33-6003-2
2. IBM System/360, PL/I Reference Manual; Form Number C28-8201
3. A Guide to PL/I - S.V. Pollock and T.D. Sterling-Holt, Reinhart and Winston, 1969.
4. NPL: Highlights of a New Programming Language - George Radin and H. Paul, Rogoway, CACM 8, 1965, p. 9.
5. Automatic Syntactic Analysis - W.M. Foster, MacDonald and Elsevair Inc.
6. Compiling Techniques, F.R.A. Hopgood, MacDonald and Elsevair Inc.
7. Anatomy of a Compiler - John A.N. Lee, Reinhold Publication Group, New York, 1967.
8. Computer Programming and Computer Systems - Anthony Hassit, Academic Press, 1967.
9. ALGOL-60 Implementation - B. Randell and L.J. Russel, Academic Press, 1964.
10. Translation of ALGOL-60 - Grau, Hill and Lambark, Springer-Verlag, New York, 1967.
11. A Multipass Translation Scheme for ALGOL-60, Hawkins and Huxtable, Annual Review in Computer Programming-3, Paragon Press.
12. WATFOR Documentation - University of Waterloo.
13. Programming Systems and Languages - Saul Rosen, McGraw-Hill 1967.
14. Scatter Storage Techniques - Robert Morris, CACM Vol. 11 p.38 (Jan. 1968).
15. Syntactic Analysis and Operator Precedence - R.W. Floyd, JACM-10, 1963, p. 316.
16. An Algorithm for Coding Efficient Arithmetic Operations - R.W. Floyd, CACM4, 1961, p. 42.

17. Sequential Formula Translation , K. Samelson and F.L.Baur, CACM3, 1960, p. 76.
18. High Speed Compilation of Object Code - C.W.Gear, CACM,8, 1965, p. 483.
19. Peephole Optimization by W.M. McKeeman, CACM 8, 1965,p.443.
20. Recursive Processes and Algol Translation, A.A.Grau, CACM4, 1961, p. 10.
21. Bounded Context Translation - Robert M. Graham, MIT, Computer Centre Report.
22. Syntax Directed Compiling, T.E. Cheatham and Sattley K., Proc. of SJCC 1964, Washington D.C.
23. Data Directed Input-Output in FORTRAN-CACM 10,1967,p.35.
24. System Programmers Guide IBM Form No. C-28-6339-5.
25. Input Output Control System IBM Form No. C-29-6309-4.
26. Programmers Guide, IBM Form No. C-28-6318.

EE-1972-M-GUP-PRO